Fachhochschule Aachen

Campus Jülich

Fachbereich: Medizintechnik und Technomathematik
Studiengang: Technomathematik

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Konzeption und Entwicklung einer sicheren, webbasierten Datenaustauschlösung im HPC-Umfeld

Masterarbeit von Martin Lischewski
Matrikelnummer: 844813

Jülich, den 26. Januar 2016

Diese Arbeit wurde betreut von:

1. Betreuer: Prof. Dr. rer. nat. Volker Sander

2. Betreuer: Dr. Bernd Schuller

Diese Arbeit ist von mir selbständig angefertigt und verfasst. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

Ort, Datum	Martin Lischewski

Vorwort

Im wissenschaftlichen Umfeld gibt es viele internationale Projektgruppen, die das High Performance Computing (*HPC*) zum Berechnen ihrer Simulationen verwenden. Im Zuge dieser Berechnungen entstehen enorme Datenmengen, die den Projektmitgliedern weltweit zur Verfügung gestellt werden müssen. Diese möchten ihre Daten auch miteinander austauschen. Um das zu gewährleisten, wird im Rahmen dieser Masterarbeit eine sichere, webbasierte Datenaustauschlösung im HPC-Umfeld konzipiert und entwickelt.
Zunächst wird das Projektumfeld und die Problematik erläutert. In Kapitel 2 werden dann die Anwendungsfälle der neuen Webapplikation definiert. Darauf aufbauend werden die Anforderungen präzisiert und in Soll- und Kannkriterien unterteilt.
Anschließend werden in Kapitel 3 die grundlegenden Technologien vorgestellt, welche im Laufe der Arbeit zum Einsatz kommen.
In Kapitel 4 wird die Applikation dann konzipiert. Die wesentlichen Funktionen werden definiert, sowie die Schnittstellen und Klassen definiert.
Das Programmkonzept wird in Kapitel 5 realisiert. Es werden Teile des Programmcodes dargestellt, beschrieben und die Benutzeroberfläche der fertigen Applikation vorgestellt.
Danach wird in Kapitel 6 die entwickelte Webapplikation evaluiert. Hierzu werden die Anforderungskriterien erneut aufgegriffen und überprüft, ob sie erfüllt sind. Zum Schluss wird die Arbeit zusammengefasst und ein kurzer Ausblick auf den weiteren Verlauf des Projektes gegeben.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Projektumfeld

Im Forschungszentrum Jülich (*FZJ*) werden Schlüsseltechnologien des 21. Jahrhunderts innovativ vorangebracht. Dazu gehören Simulationswissenschaften, Forschung mit Neutronen, Biotechnologie und viele weitere [6]. Für all diese Fachgebiete müssen oft hoch komplexe Probleme mit rechenintensiven Simulationsrechnungen gelöst werden.

Neben den wissenschaftlichen Instituten besitzt das *FZJ* ein eigenes Rechenzentrum. Das Jülich Supercomputing Centre (*JSC*) stellt die Ressourcen für wissenschaftliche Berechnungen zur Verfügung. Es betreibt am Standort Jülich mehrere High Performance Computing (HPC) Systeme wie *JUQUEEN*, *JURECA* und *JUDGE*. Das High-End System *JUQUEEN* besteht aus 28 Racks mit insgesamt 458752 Kernen und 448 TB Hauptspeicher. Es erreicht eine Rechenleistung von 5,9 Petaflops[1] und ist der zurzeit elftschnellste Supercomputer weltweit und der drittschnellste Europas [12].

Abbildung 1.1: Supercomputer JUQUEEN

[1] **FL**oating point **O**perations **P**er **S**econd (flops) steht für Fließkommaoperationen pro Sekunde. 1 Petaflops = 10^{15} flops

Die hohe Rechenleistung kann nur sinnvoll genutzt werden, wenn alle Komponenten perfekt aufeinander abgestimmt sind. Das bedeutet unter anderem, dass Programme ihre Daten schnell genug von einem externen Speicher lesen und auch ihre Ergebnisse dorthin schreiben können müssen. Die Dateisysteme, welche die Supercomputer bedienen, werden von einem separaten Cluster namens *JUST* (**JU**elich **ST**orage Cluster) bereitgestellt. Die bei den Simulationen entstehenden Datenmengen sind enorm groß und dafür bieten die Dateisysteme des *JSC* mehr als 21 Petabyte an Datenkapazität. Zusätzlich zu diesem Festplattenspeicher werden noch ca. 100 Petabyte Datenkapazität auf Bändern zur Verfügung gestellt. Die langsameren aber billigeren Bandspeicher werden zur Datensicherung und zur Datenarchivierung genutzt.

Die Arbeiten auf den *HPC*-Systemen geschehen immer projektorientiert. Wissenschaftler, welche Rechenzeit in Anspruch nehmen möchten, müssen zunächst einen Projektantrag stellen. Die Rechenzeit wird nicht wahllos vergeben, sondern ist an bestimmte Bedingungen gebunden. Unter anderem muss hierbei der wissenschaftliche Anspruch begründet werden. Die *HPC*-Systeme werden von mehreren Institutionen finanziert, deswegen wird ein entsprechend umfangreicher Genehmigungsprozess angestoßen. Nach erfolgreicher Freigabe wird ein zeitlich begrenzter Zugang bereitgestellt. Insgesamt sind für die Jülich HPC-Systeme zur Zeit mehrere 1000 Nutzer in ca. 260 Projekten registriert.

Ein Beispiel für ein Projekt, welches sehr viel Speicherplatz benötigt, ist das **H**uman **B**rain **P**roject (*HBP*) [1], ein europäisches Projekt, das ein besseres Verständnis des menschlichen Gehirns schaffen möchte. Dadurch sollen Krankheiten des Gehirns besser erkannt werden und auch Technologien entwickelt werden, die die Strukturen des Gehirns optimal nachbilden. Zur Analyse werden u.a. Bilder von einzelnen Schichten eines Gehirns aufgenommen. Ein Gehirn wird hierfür in ca. 2500 Schichten unterteilt. Hierbei werden auf Grund der Detailtreue 3500 Bilder pro Schicht erstellt. Die Bilder besitzen eine Pixelgröße von $1{,}3\,\mu m * 1{,}3\,\mu m$, so entstehen pro Schicht ca. 500 GB an Rohdaten. Hieraus folgt, dass pro Gehirn ca. $2500 * 500\,GB = 1{,}25\,PB$ Daten entstehen. Bis heute wurde ein komplettes Gehirn aufgenommen. Bis zum Jahr 2018 soll mindestens noch ein zweites Gehirn dazu kommen, das heißt, die erforderliche Datenkapazität wird sich allein für dieses Projekt mindestens verdoppeln.

Um auf die Daten zuzugreifen entwickelt das JSC, neben der Bereitstellung und Weiterentwicklung der Hardware, auch verschiedene Softwarelösungen für verteiltes und föderiertes Rechnen (Grid-Computing[2]). Diese bieten dem Nutzer Zugriff auf die verschiedensten parallelen und verteilten Systeme, welche auf verschiedensten Ressourcen basieren (z.B. HPC-Systeme, parallele Dateisysteme oder Cloud) (näheres in Kapitel 3.1).

[2]Das Grid-Computing wird verwendet um rechenintensive Probleme auf verteilten Ressourcen zu berechnen. Hierfür wird eine besondere Middleware bereitgestellt, um die Rechenleistung auf die Ressourcen zu verteilen [9].

1.2 Infrastruktur

Die Dateisysteme im Jülich Supercomputing Centre basieren auf dem von *IBM* (**I**nternational **B**usiness **M**achines Corporation) entwickelten „**G**eneral **P**arallel **F**ile **S**ystem“ kurz *GPFS* [3]. Dies ist ein speziell für Cluster entwickeltes *POSIX*[3] Dateisystem, welches mehreren Rechnern den parallelen Zugriff auf einen gemeinsamen Speicher zur Verfügung stellt. Das System ist hoch skalierbar und zugleich sehr zuverlässig.

Hardwareseitig kommen die von *IBM* entwickelten *GPFS Storage Server* (*GSS*) zum Einsatz. Dabei handelt es sich um so genannte *Building Blocks*, die aus zwei Servern bestehen. Diese Server sind wiederum an vier (*GSS 24*), bzw. sechs (*GSS 26*) *JBOD*[4] mit je 58 Festplatten angeschlossen. Ein *Building Block* besitzt keine eigenen Hardware RAID-Controller, sondern diese Funktionalität ist in eine Software-Schicht des *GPFS* in die Server Systeme integriert. Dies spart Ressourcen und ermöglicht eine effizientere Datenspeicherung. Des Weiteren bietet das *GSS* Ende-zu-Ende Prüfsummen, diese werden vor dem Schreiben der Daten gebildet und beim Lesen überprüft. Hierdurch können Festplattenfehler und Fehler, die während der Datenübertragung entstanden sind, sogenannte „silent-bit-error“, erkannt und direkt behoben werden. *JUST* besteht aus 31 *GSS 24* und zwei *GSS 26 Building Blocks*. Insgesamt wird eine Speicherkapazität von über 20 PB zur Verfügung gestellt.

Die von *JUST* bereitgestellten Dateisysteme sind auf den verschiedenen HPC-Systemen eingebunden. Insgesamt sind derzeit zwölf verschiedene Systeme an *JUST* angeschlossen. Der Supercomputer *JUQUEEN* kann auf Grund des optimierten Netzwerkanschlusses mit einer maximalen Bandbreite von bis zu 220 GB/s auf *JUST* zugreifen (siehe Abbildung 1.2).

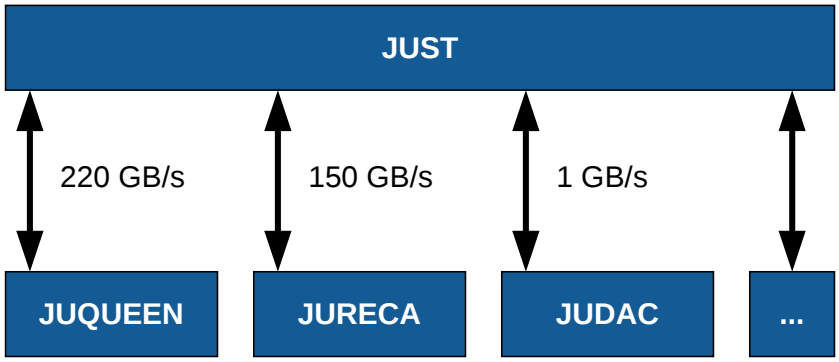

Abbildung 1.2: Anbindung zu *JUST*

[3] POSIX steht für **P**ortable **O**perating **S**ystem **I**nterface. POSIX stellt eine von *IEEE* entwickelte standardisierte Schnittstelle zwischen den Applikationen und dem Cluster dar.

[4] Der Begriff *JBOD* steht für „**J**ust a **B**unch **O**f **D**isks“. Ein JBOD ist eine „Menge“ von Festplatten, auf die einzeln zugegriffen werden kann. Im Gegensatz zu RAID-Systemen fehlt den JBOD jedoch die Redundanz.

Nutzer, die zum Beispiel per *SSH*[5] auf einem Supercomputer angemeldet sind, können auf die Daten der Dateisysteme zugreifen. Im HPC-Umfeld spricht man meist von mehreren 100 Nutzern.

1.2.1 Problematik der Infrastruktur

Das Design des Storage-Cluster *JUST* ist auf den Zugriff durch die HPC-Systeme ausgelegt. Die komplexe Architektur kann den Zugriff auf die Daten erschweren. In der Vergangenheit konnten die Nutzer nur über die HPC-Systeme auf ihre Daten zugreifen. Das bedeutet, die Nutzer waren abhängig von dem Status der HPC-Systeme. Wenn ein Supercomputer nicht zugreifbar war, z.B. während einer Wartung, oder mangels benötigter Berechtigungen auf den HPC-Systemen, konnte der Nutzer nicht auf seine Dateien zugreifen.

Ein weiteres Problem war, dass die Nutzer nur über die Login-Knoten der Supercomputer auf die Dateien zugreifen konnten, und somit die Last der Login-Knoten extrem anstieg. Die Login-Knoten sind nur für Interaktionen mit den Supercomputern ausgelegt, um z.B. auf ihnen Jobs zu starten. Sie sind allerdings nicht für den Transfer solcher Datenmengen ausgelegt, somit kam es hier zu Engpässen.

Außerdem gibt es Nutzer, die momentan keinen Zugriff auf ein HPC-System haben, deren Daten aber trotzdem auf dem *JUST* liegen. Das liegt daran, dass der Zugriff auf die HPC-Systeme für die Nutzer meistens zeitlich beschränkt ist. Die Nutzer möchten oft aber über diesen Zeitraum hinweg auf ihre Daten zugreifen, um z.B. die Daten auf ihren eigenen Speicher zu transferieren. Oft ist es für den Nutzer jedoch nicht praktikabel die produzierten, riesigen Datenmengen von *JUST* auf ein anderes Dateisystem zu transferieren. Somit benötigen Nutzer, die kein Benutzerkonto auf einem HPC-System besitzen, Zugriff auf ihre Daten im *JUST*.

1.2.2 Erste Abhilfe durch *JUDAC*

Im September 2014 wurde ein Server namens *JUDAC* (**JU**elich **D**ata **AC**cess) aufgesetzt. Dieser Server ermöglicht Nutzern einen direkten Zugriff auf alle Dateisysteme des *JUST*, ohne den Umweg über die Login-Knoten der HPC-Systeme. Um eine große Bandbreite für den Datentransfer zu gewährleisten, ist *JUDAC* über ein 10-Gigabit-Ethernet mit *JUST* verbunden. Dadurch können Daten mit ca. 1 GB/s gelesen bzw. geschrieben werden, ohne dass Engpässe wie bei den Login-Knoten auftreten (siehe Abbildung 1.2). *JUDAC* besitzt zwei *Intel Xeon* Prozessoren mit jeweils sechs Kernen und 2,66 GHz, sowie 48 GB Arbeitsspeicher. Als Betriebssystem wurde *Scientific Linux*[6] gewählt.

[5] *SSH* (**S**ecure **SH**ell) ist eine Anwendung die eine verschlüsselte Verbindung zwischen zwei vernetzten Rechnern herstellt.

[6] *Scientific Linux* ist eine durch das amerikanische Forschungszentrum *Fermi National Accelerator Laboratory* gestützte Linux Distribution, welche speziell für den wissenschaftlichen Bereich entwickelt wurde.

Alle Nutzer der Supercomputer haben automatisch auch Zugriff auf *JUDAC*. Das System läuft unabhängig von den HPC-Systemen und ist über eine 10-Gigabit-Ethernet Leitung auch von außerhalb erreichbar. Ein auf *JUDAC* angemeldeter Nutzer kann auf alle seine Dateien zugreifen. Damit besteht jetzt die Möglichkeit, Nutzern Zugriff auf die Dateisysteme zu gewähren, ohne ihnen Zugriff auf die Supercomputer geben zu müssen. Dies hat den Vorteil, da bei der Nutzung der Supercomputer die Benutzerkonten nur zeitlich begrenzt zur Verfügung stehen. Hingegen können die Benutzerkonten von *JUDAC* längerfristig eingerichtet bleiben.

Die Daten können mittels *SCP*[7] zwischen *JUDAC* und einem entfernten Rechner transferiert werden. Zusätzlich ist auf *JUDAC* einige Software des Grid-Computing, wie z.B. *UNICORE* (siehe Kapitel 3.1), vorinstalliert.

1.3 Motivation

Die Nutzer der HPC-Systeme des *JSC* sind nicht nur Mitarbeiter aus Instituten des Forschungszentrums, sondern sind weltweit verteilt. Die Forschungsprojekte sind nicht allein für die Bundesrepublik Deutschland von Interesse wie z.B. das *HBP,* welches europaweit organisiert wird. Diverse Themen haben sogar eine gewisse globale Wichtigkeit. Es schließen sich Projektgruppen weltweit zusammen, um ihre Kompetenzen auszutauschen. Mitglieder dieser Projektgruppen müssen im Rahmen ihrer Projekte natürlich auch gemeinsam auf die Ergebnisse zugreifen, um optimal mitarbeiten zu können.

Über den zentralen Gateway namens *JUDAC* können alle Nutzer des *JSC* per *SSH* auf die Dateisysteme zugreifen. Jedoch existieren bisher noch keine grafischen Oberflächen, um den Zugriff für den Nutzer zu erleichtern. Zusätzlich steigt zur Zeit das Interesse der Nutzer, Daten mit anderen Mitgliedern ihrer Projektgruppe auszutauschen. Einerseits möchten Nutzer anderen Nutzern große Datenmengen zur Verfügung stellen, andererseits möchten sie große Dateien erhalten. Der Datenaustausch könnte hierbei auch über Projektgrenzen hinweg möglich sein, also insbesondere auch mit Personen, die keinen eigenes Benutzerkonto im *JSC* haben.

Es wäre hierbei nicht praktikabel, den externen Nutzern ein temporäres Benutzerkonto einzurichten, da zu viele Konten entstehen würden. Außerdem können Benutzerkonten nicht ad hoc eingerichtet werden, da jedesmal der umfangreiche Genehmigungsprozess durchlaufen werden müsste. Der administrative Aufwand wäre zudem sehr hoch, da spezifische Regeln eingehalten werden müssten, um die Sicherheit der Dateisysteme zu gewährleisten.

Bisher gestaltete sich der Zugriff solcher externer Nutzer auf die Daten äußerst schwierig. Als eine mögliche Lösung wurde ein *FTP*-Server (siehe Kapitel 3.2) aufgesetzt. Die Nutzer konnten ihre Daten auf den *FTP*-Server laden und dort anderen Personen zur Verfügung stellen. Diese Lösung ist jedoch nicht praktikabel, da ein *FTP*-Server deutliche Sicherheits-

[7] *SCP* (**S**ecure **C**o**P**y) ist eine Anwendung mit der Daten verschlüsselt zwischen vernetzten Rechnern ausgetauscht werden.

lücken aufweist, was im Verlauf dieser Arbeit beschrieben wird. Durch einen Wildwuchs von *FTP*-Servern werden beliebig viele Angriffsziele durch offene Ports generiert. Außerdem müssen für den *FTP*-Server zusätzliche Speicherkapazitäten zur Verfügung gestellt werden, da der *FTP*-Server meist unabhängig vom *GPFS* läuft. Weiterhin kann ein öffentlicher *FTP*-Server zu rechtlichen Problemen führen, da die externen Nutzer das System zum illegalen Datenaustausch missbrauchen könnten.

Da bisher der Zugriff externer Mitarbeiter auf die Daten nicht eindeutig geregelt ist, besteht zusätzlich die Gefahr, dass die Daten einfach bei *Dropbox*[8] oder anderen Cloud Anbietern hochgeladen werden. Hierbei verliert man jedoch den Besitz der Daten, da kostenlose Cloud-Anbieter stets den Besitz der von ihnen gespeicherten Daten beanspruchen. Bei einer kostenpflichtigen Lösung verliert man zwar nicht unbedingt den Besitz der Daten, allerdings erhält man auch hierbei keine Kontrolle, wo sich die Daten genau befinden und wer entsprechend Zugriffsrechte darauf hat. Die Auslagerung der Dateien in Cloud-Dienste ist stets mit Kosten verbunden, welche bei den großen Datenmengen, wie sie durch die HPC Systeme entstehen, schnell unüberschaubar werden. Für diese riesigen Datenmengen ist weder die Datenanbindung des Forschungszentrums noch die diverser Cloud-Anbieter ausgelegt, so dass die Übertragung recht lange dauert.

Das *JSC* bevorzugt eine Lösung, bei welcher die Daten auf dem lokalen *JUST* liegen bleiben. Dadurch entsteht zum einen kein zusätzlicher Datenverkehr durch Synchronisation der Server untereinander, zum anderen bleibt der Besitz der Daten beim *JSC*. Außerdem wird kein weiterer Storage benötigt.

Abbildung 1.3: Juelich Storage Cluster (*JUST*)

Die Lösung für die genannten Probleme ist eine Anwendung, welche die Daten für externe Nutzer leicht erreichbar macht. Hierzu gibt es schon Softwarelösungen, wie zum Beispiel

[8] *Dropbox* ist eine kommerzielle, Cloud-basierte Softwarelösung für das Verwalten und den Austausch von Dateien.

die Open-Source Software *ownCloud*[9]. Der Nachteil dieser Lösungen ist aber, dass sie nicht vertrauenswürdig sind, zum Beispiel weil der Webserver die Daten besitzen muss. Solche Cloud-Lösungen benötigen ein eigenes Benutzerkonto auf dem Dateisystem, unter dem die Daten abgelegt werden. Dadurch wird die Datei nicht dem richtigen Besitzer bzw. der richtigen Gruppe zugeteilt. *JUST* dagegen sollte nicht angepasst werden, sondern die Daten sollen einfach dort liegen bleiben, so dass keine Sicherheitsrichtlinien geändert werden müssen. Der Webserver wird also nicht Besitzer der Dateien, sondern er greift nur durch das interne Netz auf die Dateisysteme zu. Der Webserver und *JUST* laufen getrennt voneinander und das Dateisystem kennt den Webserver nicht.

Weiterhin bieten vorgefertigte Softwarelösungen oft überflüssige Funktionen, die nicht benötigt werden, oder bergen Sicherheitsrisiken, wie zum Beispiel das offline verfügbar machen von Verzeichnissen oder Dateien, sowie die dadurch entstehende Synchronisation bei Aktualisierungen.

Gerade in der heutigen Zeit, in der immer wieder geplante und ungeplante Sicherheitslücken in Programmen auftauchen, welche entweder durch fehlerhafte Programmierung, oder durch Behörden in den Quellcode gelangt sind, ist es bei sensiblen Daten wichtig, dass man alle verwendeten Komponenten sowie Protokolle genauestens kennt, um bei Angriffen von außen entgegen wirken zu können. Eine selbst entwickelte Applikation kann bei Berücksichtigung all dieser Punkte einen vertrauenswürdigen und benutzerfreundlichen Zugriff auf die internen Dateien des Forschungszentrums gewährleisten.

Zur Lösung dieser Problematik wird im Rahmen dieser Masterarbeit nun eine sichere, webbasierte Datenaustauschlösung im HPC-Umfeld konzipiert und entwickelt.

[9] *ownCloud ist* eine Open-Source, Cloud-basierte Softwarelösung für das Verwalten und den Austausch von Dateien.

2 Aufgabenstellung

2.1 Anwendungsfälle

In diesem Kapitel werden mögliche Szenarien zur Nutzung der Applikation beschrieben. Die folgende Abbildung 2.1 stellt den ersten Anwendungsfall vereinfacht dar.

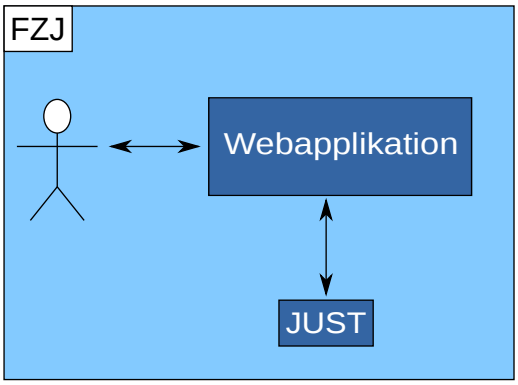

Abbildung 2.1: Anwendungsfall 1: Interner Zugriff

Das hellblaue Rechteck stellt das interne Netz des *FZJ* dar. Die Webapplikation soll innerhalb dieses Netzes laufen und von dort aus direkt auf *JUST* zugreifen.

Ein Nutzer greift aus dem internen Netz auf die Applikation zu, dies wird in Abbildung 2.1 durch das Strichmännchen innerhalb des hellblauen Rechtecks gekennzeichnet. Der Nutzer besitzt bereits ein *LDAP*-Benutzerkonto und greift über die Webapplikation auf seine Daten zu. Hierfür muss er sich zunächst authentisieren und bekommt anschließend alle seine Dateien aufgelistet. Er kann nun seine Dateien von *JUST* runterladen, umbenennen, verschieben und löschen, sowie neue Dateien hochladen.

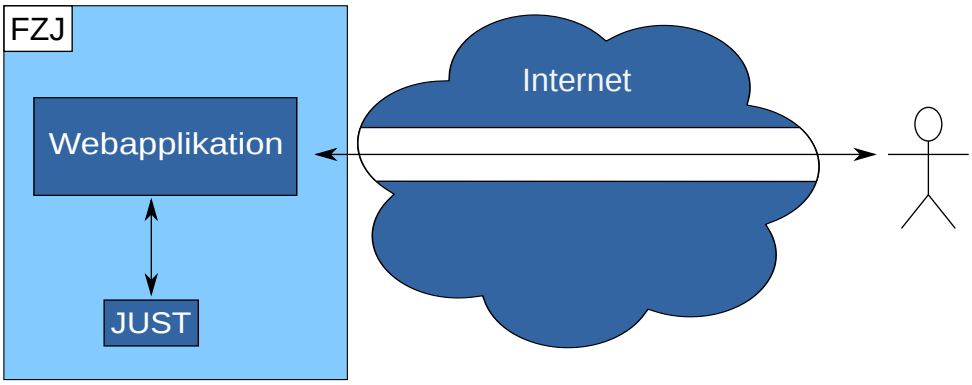

Abbildung 2.2: Anwendungsfall 2: Externer Zugriff

Im zweiten Anwendungsfall (siehe Abbildung 2.2) greift nun ein Nutzer über das Internet auf seine Dateien zu. Dieser könnte ein Mitarbeiter des *FZJ* sein, der an Projektergebnissen von anderen Standorten arbeiten möchte. Andererseits gibt es Nutzer von *JUST*, die nicht Mitarbeiter des *FZJ* sind und somit von außerhalb auf ihre Daten zugreifen müssen. Ein Nutzer kann also auch von außerhalb, weltweit auf seine Dateien zugreifen. Der Zugriff muss hierbei über eine gesicherte Verbindung geschehen.

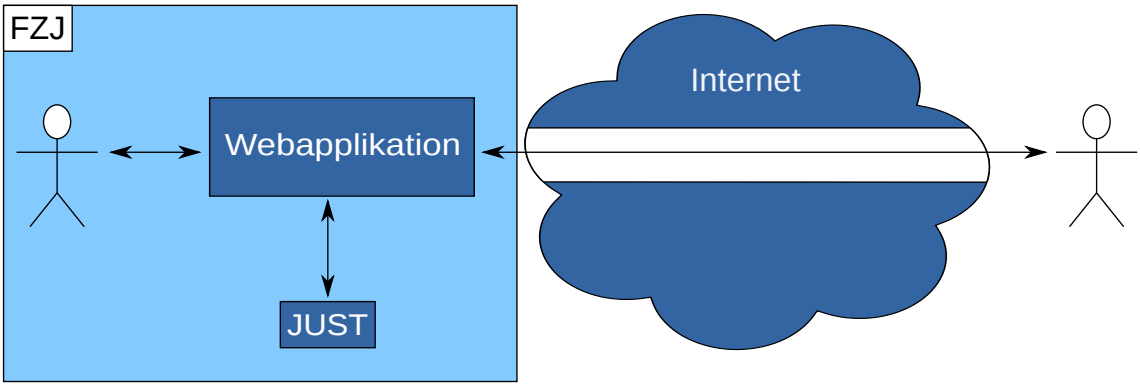

Abbildung 2.3: Anwendungsfall 3: Datenaustausch

Abbildung 2.3 zeigt einen Nutzer, der seine Dateien einem anderen Nutzer zur Verfügung stellt. Das bedeutet, er kann anderen Nutzern das Recht geben auf seine Dateien lesend oder sogar schreibend zuzugreifen. Freigaben können auch für ganze Verzeichnisse gelten. Diese Freigaben können auch verändert oder aufgehoben werden. Jeder angemeldete Nutzer soll zu seinen eigenen Dateien auch all die Dateien angezeigt bekommen, die ihm von anderen Nutzern freigegeben wurden. Die Freigabe ist nun sowohl für interne als auch externe Nutzer möglich.

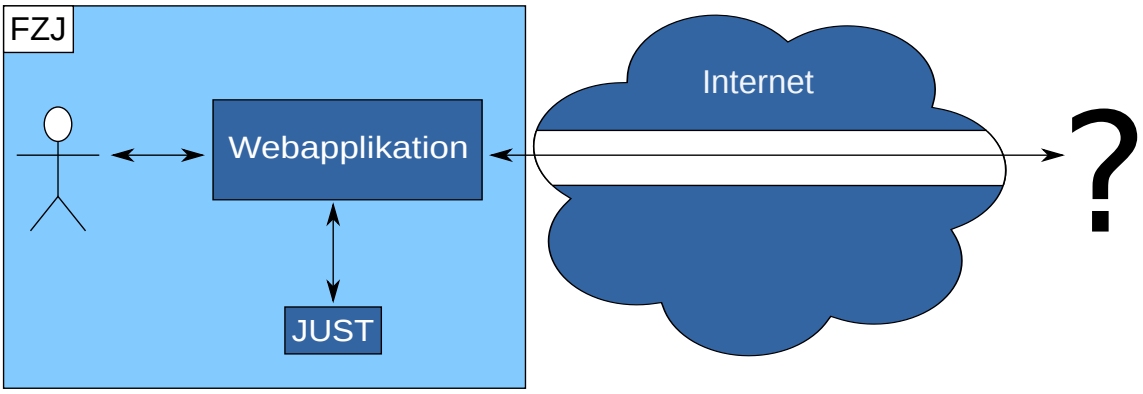

Abbildung 2.4: Anwendungsfall 4: Datenaustausch mit unbekannten Nutzern

Optional kann die Webapplikation nicht nur Nutzern, welche bereits ein *LDAP*-Benutzerkonto besitzen, dienen, sondern die Daten können auch mit anderen Projektpartnern ausgetauscht werden. Diese Projektpartner, für die ad hoc ein Nutzerkonto eingerichtet wird, sind zunächst für die Applikation unbekannt und werden hier durch das Fragezeichen dargestellt. Die Authentisierung erfolgt hierbei nach dem gleichen Muster wie bei den anderen Fällen. Anschließend bekommen die Nutzer Dateien angezeigt, welche ihnen freigegeben wurden.

2.2 Anforderungen an die Applikation

Folgende Sollkriterien ergeben sich aus den Anwendungsfällen und werden zwingend an die Applikation gestellt:

SK1: Jeder Nutzer von *JUST* soll Zugriff auf seine Daten erhalten.

SK2: Ein Nutzer soll anderen Nutzern Zugriff auf seine Dateien gewähren können.

SK3: Der Besitz der Daten darf nicht an Dritte abgegeben werden.

SK4: Die Struktur der Dateisysteme soll nicht geändert werden.

SK5: Die Applikation soll sicher sein und ein vertrauenswürdiges Umfeld bieten.

SK6: Die Daten sollen nicht auf ein weiteres Dateisystem transferiert werden müssen.

SK7: Sicherheitsrichtlinien sollen nicht geändert werden.

SK8: Die Applikation soll sehr flexibel erreichbar sein.

Optional ergeben sich folgende Kannkriterien, welche die Applikation nach Möglichkeit erfüllen soll:

KK1: Die Applikation soll möglichst einfach und intuitiv bedienbar sein.

KK2: Nutzer müssen nicht zwingend ein Benutzerkonto auf den HPC-Systemen besitzen.

KK3: Die Applikation soll leicht erweiterbar sein.

3 Grundlagen

In diesem Kapitel werden die grundsätzlichen Technologien sowie Fachbegriffe erläutert, welche im weiteren Verlauf der Arbeit immer wieder aufgegriffen werden. Zudem wird auf die Vor- und Nachteile der einzelnen Technologien eingegangen und deren Unterschiede hervorgehoben.

3.1 Middleware *UNICORE*

UNICORE (**UNI**form Interface to **CO**mputing **RE**sources) ist eine vom *JSC* mitentwickelte Middleware für das Grid-Computing und kommt heute in vielen Supercomputer-Zentren zum Einsatz. *UNICORE* stellt Software zur Verfügung, die das verteilte Rechnen und das Nutzen von verteilten Ressourcen leichter und sicherer macht [13].

Das Softwarepaket ist ein sehr mächtiges Tool. Der Nutzer kann unter anderem Jobs absetzen, Workflows definieren, auf Dateisysteme zugreifen und Daten transferieren. Das Tool dient auch als Gateway und kann Webservices routen und Firewalls einrichten.

Die Technologie ist Open-Source unter der *BSD*-Lizenz[1]. Die Plattformunabhängigkeit wurde durch eine Implementierung in *Java* erreicht. *UNICORE* unterstützt viele verschiedene Batch- und Betriebssysteme wie zum Beispiel *Windows*, *MacOS, Linux*, *SLURM* und *Torque*. Es ist schnell und leicht zu installieren und zu konfigurieren.

Der Leitgedanke ist eine „Service- Oriented Architecture“ (*SOA*) zu entwickeln. Die gesamte Software soll einfach erweiterbar sein und einzelne Komponenten sollen leicht austauschbar sein. Das Design basiert auf klaren und strikten Leitprinzipien, die der leichten Erweiterbarkeit dienen. Da die Architektur und die Schnittstellen auf Standards basieren, ist die Zusammenarbeit mit anderen Grid-Technologien gewährleistet. Weiterhin bietet *UNICORE* Mechanismen, um verschiedene Applikationen aus wissenschaftlichen oder industriellen Bereichen zu integrieren.

UNICORE beinhaltet verschiedene Benutzerschnittstellen für den Zugriff auf Ressourcen. Diese sind sicher, intuitiv und modern. Es gibt drei verschiedene Clients, welche den Bedürfnissen verschiedenster wissenschaftlicher Gemeinschaften angepasst sind. Dazu gehören ein auf *Eclipse* basierender, grafischer Client, ein Client in Form einer Kommandozeile und

[1] Die *BSD*-Lizenz (**B**erkley **S**oftware **D**istribution) ist eine Open-Source Lizenz von der University of California, Berkeley.

ein Webportal.

Die Workflows können über Graphen, Schleifen und Bedingungen definiert werden. Die Workflow Sprache ist hierbei frei wählbar.

UNICORE bietet eine sichere Authentisierung und Autorisierung über offene Standards wie zum Beispiel *TLS* und *X.509* Zertifikate.

3.2 Dateitransferprotokoll *FTP*

FTP (**F**ile **T**ransfer **P**rotocol) ist ein Protokoll zur Datenübertragung [10]. *FTP* ermöglicht dem Nutzer Daten an einen entfernten Rechner zu senden, oder Daten von ihm zu empfangen. Dateien werden seriell übertragen. Zusätzlich können entfernte Verzeichnisse erstellt, ausgelesen und gelöscht werden. Ein *FTP*-Client ist auf Rechnersystemen verschiedenster Art verfügbar.

Das Protokoll wird der Anwendungsschicht des *TCP/IP*-Referenzmodells zugeordnet, dies entspricht den Schichten fünf bis sieben im *ISO/OSI*-Referenzmodell. *TCP* (Transmission Control Protocol) wird bei *FTP* als Protokoll der Transportschicht verwendet. *TCP* ist verbindungsorientiert und gewährleistet die Zuverlässigkeit der Datenübertragung. Das bedeutet, dass fehlerhafte oder gar nicht empfangene Datenpakete neu angefordert werden. Auf der Internetschicht kommt das *IP* (Internet Protocol) zum Einsatz, welches sich die entsprechenden Wege durch die unterschiedlichen Router selbständig sucht.

FTP arbeitet nach dem Client-Server Prinzip. Das bedeutet, es muss immer einen *FTP*-Server geben, auf dem die entsprechenden Daten abgelegt werden. Außerdem müssen entsprechende Berechtigungen und Freigaben gesetzt werden. Auf der Gegenseite muss auf dem System ein *FTP*-Client verfügbar sein, welcher sich entsprechend am *FTP*-Server authentisiert und somit auf die Daten zugreifen kann. Die Authentisierung geschieht hierbei unverschlüsselt.

Der Datenaustausch geschieht über zwei getrennte *TCP*-Verbindungen. Es gibt eine Steuerverbindung, über die Befehle und entsprechende Antworten versendet werden. Die eigentlichen Daten werden jedoch über eine Datenübertragungsverbindung geschickt.

Es werden vier mögliche Datentypen definiert. Einerseits *ASCII* (American Standard Code for Information Interchange) und *EBCDIC* (Extended Binary Coded Decimals Interchange Code), die zumeist für Textdateien verwendet werden. Andererseits *Image* und *Logical Octet Size* die für binäre und verschlüsselte Daten verwendet werden.

FTP definiert drei verschiedene Übertragungsmodi

- Stream
  Bei diesem Übertragungsmodus werden die Daten ohne besondere Verfahren versendet.

- Block
  Bei diesem Übertragungsmodus werden die Daten in Blöcke eingeteilt. Die Blöcke werden durchnummeriert, so können Sender und Empfänger den Fortschritt der Übertragung überwachen. Falls die Verbindung getrennt wird, kann die Übertragung später beim letzten fehlerfrei übertragenen Block wieder beginnen.

- Compressed
  Bei diesem Übertragungsmodus werden die Daten vor dem Senden komprimiert. Der Empfänger muss die Daten anschließend wieder dekomprimieren.

3.3 Dateitransfer mittels *UFTP*

UFTP (***U****NICORE* ***FTP***) ist eine Entwicklung des *JSC*. Es ist ein Transportprotokoll basierend auf *FTP*. Gleichzeitig bietet es eine *Java* Bibliothek zum Datenaustausch [14]. Entwickelt wurde *UFTP* im Rahmen von *UNICORE* für das Grid-Computing, kann aber in jeglichen *UNIX*-Umgebungen eingesetzt werden.

Die Motivation für die Entwicklung von *UFTP* war der Bedarf eines sicheren, schnellen und parallelen Transportprotokolls zum Datenaustausch. Der Datentransfer sollte komfortabel über verschiedene Firewalls hinweg ermöglicht werden. Zudem soll das Sicherheitsrisiko minimiert werden, welches bei *FTP* durch viele offene Ports besteht.

UFTP ist fest in *UNICORE* integriert, kann aber auch als eigenständige Instanz laufen. Es ist zuständig für den Datentransfer zwischen einem *UFTP*-Server und einem *UFTP*-Client.

Das Protokoll bietet einen hochperformanten Datenaustausch zwischen einem *UFTP*-Server und einem *UFTP*-Client auf Basis des *TCP* Protokolls. Der hohe Durchsatz wird mithilfe eines pseudo-*FTP* Protokolls erreicht, bei dem dynamisch mehrere Ports geöffnet werden können. Über diese Ports können die Daten dann parallel gesendet werden. Das dynamische Öffnen der Ports hat den weiteren Vorteil, dass *UFTP* (im Gegensatz zum herkömmlichen *FTP*) standardmäßig nur einen einzelnen nach außen offenen Port benötigt. Die Datenströme können optional verschlüsselt und komprimiert werden (siehe Kapitel 3.3.6). Weiterhin bietet *UFTP* die Möglichkeit partiell lesend oder schreibend auf eine Datei zuzugreifen. Es muss nicht bei jedem Zugriff die ganze Datei ausgelesen werden, sondern es kann einzeln auf Abschnitte der Datei zugegriffen werden. Falls vom Dateisystem unterstützt, können diese Zugriffe auf die selbe Datei sogar parallelisiert werden.

UFTP besteht aus drei Instanzen, dem *UFTP*-Server, dem *UFTP*-Client und einem Authentifizierungsserver. Diese drei Instanzen werden im Folgenden genauer betrachtet.

3.3.1 *UFTP*-Server

Der *UFTP*-Server läuft als *UFTP* Daemon (*UFTPD*) direkt auf den Dateisystemen, bzw. auf einem Knoten, der diese eingebunden hat. Ein Daemon ist ein Programm das ständig im Hintergrund läuft und dem Nutzer verschiedene Dienste zur Verfügung stellt. Der *UFTPD* stellt einerseits dem *UFTP*-Client Dienste zum Datenaustausch zur Verfügung, (siehe Kapitel 3.3.3) andererseits beinhaltet er über einen abgesicherten Port Dienste zur eigenen Steuerung bzw. Kontrolle.

Um Zugriff auf das gesamte Dateisystem zu haben, läuft der *UFTPD* als `root`. Alle Dateizugriffe geschehen jedoch nur über die UID[2] bzw. GID[3] des aktuellen Nutzers.

UFTPD verwendet zwei frei wählbare Ports.
Der „command“ Port ist der gesicherte Port, über den der Daemon gesteuert wird. Er ist *SSL* geschützt, und nur durch den Authentifizierungsserver erreichbar. Dieser Port ist eine mögliche Angriffsstelle und muss deshalb besonders geschützt werden (siehe Kapitel 3.3.6). Der „listen“ Port akzeptiert Datenverbindungen vom Client. Er übernimmt die Funktion der Kontrollverbindung beim herkömmlichen *FTP*. Dieser Port muss für den Client offen sein. Es ist der einzige offene Port auf dem *UFTP*-Server. Über den „listen“ Port werden die Daten direkt vom Client empfangen bzw. gesendet.

3.3.2 Authentifizierungsserver

Der Authentifizierungsserver verwaltet die Benutzer. Er bildet den Benutzernamen, welchen er vom *UFTP*-Client erhält, auf die entsprechende UID bzw. GID ab. Meist läuft dieser Dienst auf einem separaten Server.

Der Authentifizierungsserver hat Zugriff auf den „command“ Port des *UFTP*-Servers. So kann er den *UFTPD* steuern und eine neue Datenverbindung einleiten.

3.3.3 *UFTP*-Client

Der *UFTP*-Client läuft auf dem Rechner des Endnutzers, und stellt dort verschiedene Funktionen zur Verfügung. Dateien können sowohl herunter- als auch hochgeladen werden. Weiterhin gibt es Funktionen zum Filtern und Auflisten. Dateien können kopiert, verschoben und umbenannt werden. Diese Operationen werden immer über eine direkte Verbindung an den *UFTP*-Server gesendet und dort ausgewertet (siehe Kapitel 3.3.4).

Der Client authentisiert sich gegenüber dem Authentifizierungsserver mit einem Benutzernamen und dem Passwort. Alternativ kann zur Authentisierung auch ein *SSH*-Schlüssel

[2]UID (User-ID) ist der Identifikator eines Benutzers.
[3]GID (Group-ID) ist der Identifikator einer Benutzergruppe.

genutzt werden. Somit benötigt der *UFTP*-Client Zugriff zum Authentifizierungsserver und zum *UFTPD*.

3.3.4 Aufbau einer Datenverbindung

Der Aufbau einer Datenverbindung mittels *UFTP* wird in Abbildung 3.1 dargestellt.

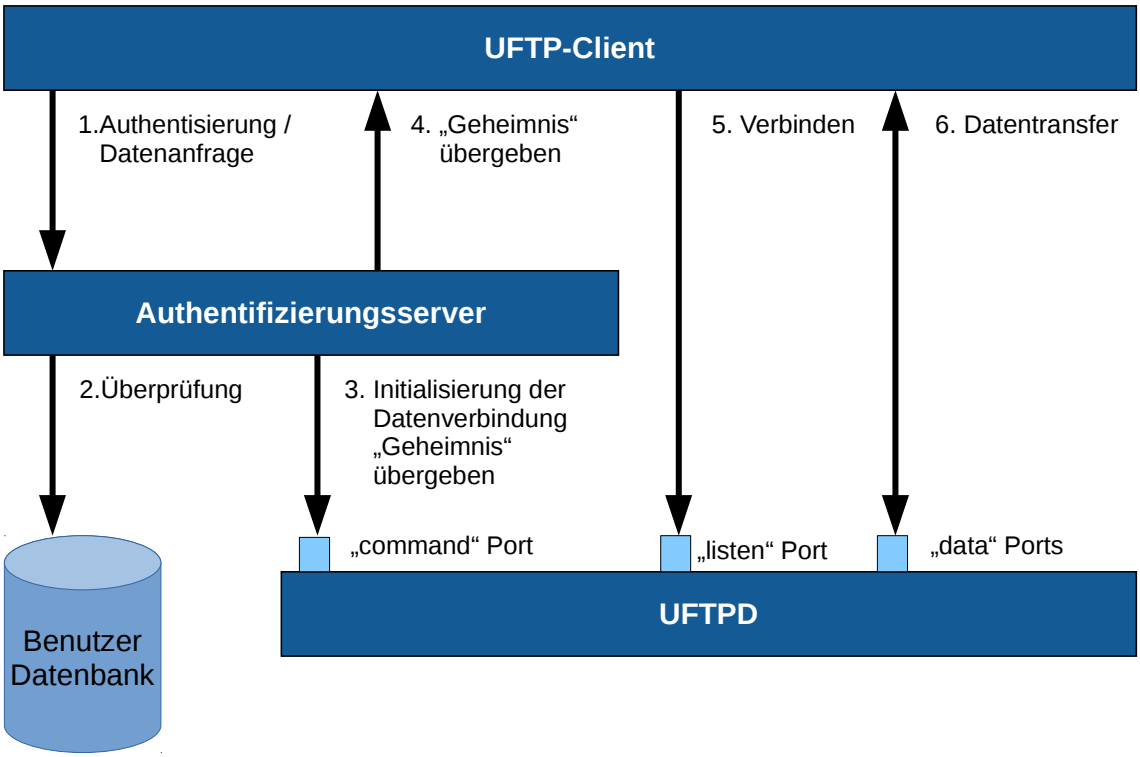

Abbildung 3.1: Aufbau einer Datenverbindung mit *UFTP*

Eine Datenverbindung wird vom *UFTP*-Client angestoßen. Als erstes authentisiert sich der Client mit Benutzername und Passwort (alternativ SSH Schlüssel) beim Authentifizierungsserver. Gleichzeitig teilt der *UFTP*-Client dem Authentifizierungsserver auch mit, auf welche Datei er lesend oder schreibend zugreifen möchte.
Im zweiten Schritt überprüft der Authentifizierungsserver die Benutzerdaten gegenüber einer Benutzerdatenbank. Hierbei wird der Benutzername auf die entsprechende Benutzer-ID bzw. Gruppen-ID abgebildet.
Im Erfolgsfall initialisiert der Authentifizierungsserver im dritten Schritt die Datenverbindung beim *UFTPD*. Hierfür schickt er eine Anfrage an den „command“ Port des *UFTP*-Servers. Diese Anfrage beinhaltet folgende Informationen:

- *IP*-Adresse des Clients
- Name der Datei

- Senden oder Empfangen
- Anzahl der Datenverbindungen
- Nutzer und Gruppen-ID
- „Geheimnis“ (siehe Kapitel 3.3.6)
- Optionaler Schlüssel zum Verschlüsseln (siehe Kapitel 3.3.6)

Die Anzahl der Datenverbindungen bestimmt, über wie viele Verbindungen der *UFTP*-Client und der *UFTP*-Server die Daten parallel austauschen werden. Das „Geheimnis“ ist zufällig generiert, und dient der späteren Authentifizierung des Clients. Ab jetzt wartet der *UFTPD* auf die Verbindung vom Client.
Im vierten Schritt muss das „Geheimnis“ nun auch dem Client bekannt gemacht werden. Anschließend verbindet der Client sich mit dem *UFTP*-Server, zur Verifizierung wird das „Geheimnis“ genutzt. Diese Verbindungsanfrage läuft über den offenen „listen“ Port. Der *UFTPD* überprüft die *IP*-Adresse und das „Geheimnis“.
Falls der Server den Client akzeptiert, werden im sechsten Schritt die Datenkanäle geöffnet. Die Daten können nun direkt zwischen dem Server und dem Client ausgetauscht werden.

3.3.5 *UFTP*-Sitzung

Standardmäßig wird der in Kapitel 3.3.4 beschriebene Ablauf bei jedem Dateizugriff durchgearbeitet. *UFTP* bietet jedoch auch die Möglichkeit, eine Nutzersitzung zu starten. Die Authentisierung des Clients geschieht einmalig beim Starten der Sitzung. Während dieser Sitzung kann der Nutzer auf alle seine Dateien zugreifen und muss sich nicht jedes mal gegenüber dem Authentifizierungsserver ausweisen.

Der Start einer Sitzung geschieht ähnlich wie der Aufbau einer einzelnen Datenverbindung mit dem Unterschied, dass der Client anstatt des Namens einer Datei eine vordefinierte Zeichenkette übergibt. Diese Zeichenkette signalisiert, dass eine Sitzung gestartet werden soll. Der Authentifizierungsserver überprüft die Authentisierung und gibt die Anfrage entsprechend an den *UFTP*-Server weiter. Der *UFTP*-Server erkennt anhand der Zeichenkette, dass eine Sitzung gestartet wird, und von nun an lässt er jede Datenverbindung vom Client zu.

Die *UFTP*-Sitzung bleibt solange bestehen, bis der Client die Verbindung beendet.

Der Vorteil einer *UFTP*-Sitzung ist, dass die Kommunikation zum Authentifizierungsserver nach der einmaligen Authentifizierung komplett wegfällt, anstatt bei jedem Dateizugriff neu durchlaufen zu werden. Der Nutzer kann dadurch viel schneller auf seine Dateien zugreifen und die Netzlast wird gesenkt. Ein Nachteil ist jedoch, dass während der gesamten Sitzung die Serverlast steigt, da die Ports offen bleiben, und ein Thread pro Sitzung benötigt wird.

3.3.6 Sicherheit in *UFTP*

FTP ist standardmäßig ein unsicheres Protokoll. Die Authentisierung funktioniert ausschließlich mit Nutzername und Passwort, welche im Klartext über den Kontrollport übertragen werden. Die Kommunikation kann somit leicht abgehört werden. Außerdem muss der Kontrollport nach außen offen sein und ist dadurch angreifbar.

UFTP fügt nun zu *FTP* noch einen sicheren Kontrollkanal hinzu, der über einen gesicherten „command“ Port läuft. Über diesen wird der Client authentisiert und der Datentransfer initialisiert. Dieser Port stellt ein besonderes Sicherheitsrisiko dar, denn sobald ein Angreifer den „command“ Port übernommen hat, kann er auf alle Daten der Dateisysteme zugreifen. Deshalb ist der „command“ Port durch *TLS* gesichert. Zusätzlich ist er, im Gegensatz zum Kontrollport von *FTP*, nicht nach außen offen, sondern durch die Firewall geschützt.

Die Authentifizierung läuft über einen eigenständigen Authentifizierungsserver. Die Anmeldedaten werden immer verschlüsselt übertragen, so dass ein möglicher Angreifer, der die Leitung abhört, die Anmeldedaten nicht verstehen kann.

Nach der Authentifizierung bekommen Client und Server ein „Geheimnis“, welches verschlüsselt übertragen wird. Dieses „Geheimnis“ ist zufällig generiert und nur über einen begrenzten Zeitraum gültig. Es authentifiziert den Client gegenüber dem Server. Beim Verbindungsaufbau überprüft der Server die Client IP und das „Geheimnis“, sodass kein anderer über diese Sockets zugreifen kann.

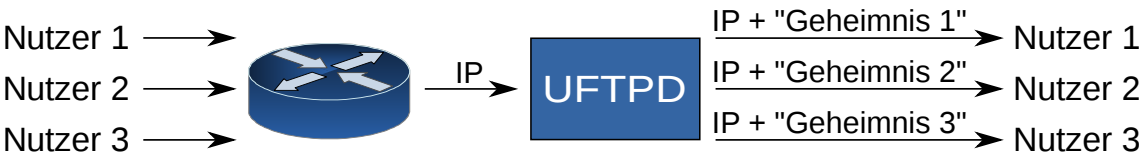

Abbildung 3.2: Zuordnung des Nutzers durch *IP* und „Geheimnis“

Die Abbildung 3.2 zeigt ein Beispiel, bei welchem drei verschiedene Nutzer über die gleiche *IP*-Adresse auf den *UFTPD* zugreifen. Zunächst kann der *UFTP*-Server die Anfragen nicht eindeutig den Nutzern zuordnen, da er nur die gleiche *IP*-Adresse von allen dreien bekommt. Durch das eindeutige „Geheimnis“ in Kombination mit der *IP*-Adresse, wird die Zuordnung jedoch möglich. Ein weiterer Fall, bei dem ohne „Geheimnis“ die Nutzer nicht eindeutig zugeordnet werden können, entsteht wenn mehrere Nutzer den selben Rechner gleichtzeitig nutzen. Oder wenn ein Nutzer mehrere Datentransferverbindungen gleichzeitig startet, können die Datenverbindungen ohne „Geheimnis“ nicht eindeutig zugeordnet werden.

Die Kommunikation zwischen Client und Server kann optional symmetrisch verschlüsselt und auch komprimiert werden. Dadurch wird eine private Verbindung gewährleistet und diese wird vor Angreifern geschützt.

Der Server läuft zwar als root, der Datenzugriff geschieht aber über die effektiven Nutzer-

IDs und Gruppen-IDs. Hierdurch wird gewährleistet, dass der Nutzer nur auf seine eigenen Daten zugreifen kann.

Das Standardprotokoll *SFTP* ist eine sicherere Alternative zu *FTP*. Im Gegensatz zu *FTP* ist der Kontrollkanal - wie bei *UFTP* - verschlüsselt und somit sicher. Allerdings bietet es Angreifern von außen mehr Angriffsmöglichkeiten als *UFTP*. Bei *SFTP* müssen -wie bei *FTP*- ganze Port-Bereiche geöffnet werden. *UFTP* benötigt jedoch standardmäßig nur einen offenen Port und kann dynamisch weitere Verbindungen öffnen, dies bietet einen enormen Sicherheitgewinn für dieses Projekt. Ein weiterer Nachteil von *SFTP* ist, dass alle Nutzer ein Benutzerkonto auf dem Server-System benötigen. Im Gegensatz dazu kann der Authentifizierungsserver von *UFTP* Nutzer auf eine beliebige *UNIX*-ID abbilden, und somit auch Nutzern, die eigentlich kein Benutzerkonto besitzen, Zugriff gewähren.

3.4 Verschlüsselte Kommunikation *TLS / SSL*

TLS (**T**ransport **L**ayer **S**ecurity) oder auch *SSL* (**S**ecure **S**ockets **L**ayer) ist ein Protokoll der Transportschicht zur kryptografischen Verschlüsselung von Netzwerkkommunikation [4]. Das Protokoll ist standardisiert und weit verbreitet.

TLS ist die Weiterentwicklung von *SSL*. Es sichert die Kommunikation zwischen einem Server und einem Client. Über das *SSL-Handshake* Protokoll werden der Server und optional auch der Client zunächst über Zertifikate beidseitig authentifiziert. Ein Zertifikat ist ein digitaler Ausweis, der die Identität einer Person oder eines Servers „ausweist“. Es beinhaltet unter anderem den öffentlichen Schlüssel des Zertifikatinhabers. Um dem Empfänger zu gewährleisten, dass das Zertifikat echt ist, also dass die Bindung zwischen dem öffentlichen Schlüssel und der Person bzw. dem Server korrekt ist, wird das Zertifikat von einer vertrauenswürdigen Zertifizierungsstelle (CA, Certification Authority) signiert.

Nach der Authentifizierung wird zwischen den beiden Kommunikationsteilnehmern der stärkste gemeinsame Algorithmus zur symmetrischen Verschlüsselung bestimmt. Der Vorteil einer symmetrischen Verschlüsselung ist, dass sie schneller als eine asymmetrische Verschlüsselung ist. Jedoch muss der Schlüssel zur Verschlüsselung zunächst bestimmt und ausgetauscht werden. Damit der Schlüssel jedoch nicht abgehört werden kann, wird dieser über asymmetrische Verschlüsselungsverfahren zwischen den Teilnehmern ausgetauscht. Anschließend kann die Verbindung über den symmetrischen Schlüssel verschlüsselt werden. Somit ist eine private Verbindung gewährleistet, bei welcher nur die zwei Kommunikationspartner die Nachrichten lesen können.

TLS kommt unter anderem bei der Kommunikation über *HTTPS* (**H**yper**T**ext **T**ransfer **P**rotocol **S**ecure) zum Einsatz. *HTTPS* bietet eine über *TLS* abgesicherte Kommunikation auf Basis von *HTTP* (**H**yper**T**ext **T**ransfer **P**rotocol). Hiermit kann die Kommunikation zwischen einem Webserver und dem Browser des Clients abgesichert werden. Dieses Protokoll kommt bei besonders sicherheitsrelevanten Anwendungen zum Einsatz, wie z.B. dem Onlinebanking, Web-Mail und bei Online-Shops.

Die Verbindung wird immer durch eine Signatur abgesichert. Durch einen verschlüsselten Hashwert wird sichergestellt, dass die Daten nicht von Dritten verändert wurden. Dadurch ist die Integrität der Daten stets gewährleistet.

3.5 *Java*-Framework *Vaadin*

Vaadin (finnisch für weibliches Rentier) ist ein *Java*-Framework zur Entwicklung von Benutzeroberflächen in Webapplikationen. Es ist Open-Source und unter der Apache-Lizenz lizenziert. Hierdurch wird sichergestellt, dass der Code weitergegeben und weiterentwickelt werden dürfen.

Vaadin läuft serverseitig in einem *Java* fähigen Webserver. Clientseitig sehen die Nutzer eine *HTML5* Webapplikation in ihrem Browser, dadurch bietet *Vaadin* automatisch Kompatibilität zu jedem herkömmlichen Browser. Neben dem Browser wird clientseitig keine weitere Software, wie z.B. eine *Java* Laufzeitumgebung, benötigt. Die Kommunikation zwischen dem Browser und dem Server geschieht über *Ajax* (Asynchronous *JavaScript* and *XML*) und wird durch *Vaadin* automatisiert.

Einer der Vorteile von *Vaadin* ist eine große Gemeinschaft von 150.000 Entwicklern [8]. Es ist außerdem leicht erweiterbar, mittlerweile wurden mehr als 500 Erweiterungen durch diese Entwickler-Vereinigung entwickelt [8]. *Vaadin* ist gut geeignet für Webapplikationen bestehend aus einer einzigen Seite, jedoch weniger gut für mehrseitige Webseiten. Durch eine klare *API*[4] ist *Vaadin* wartungsfreundlich.

3.5.1 Anwendung von *Vaadin*

Die Programmierung in *Vaadin* ähnelt sehr der herkömmlichen Anwendungsprogrammierung. Ähnlich wie *Swing*[5] bietet *Vaadin* mehrere grafische Komponenten zum Erstellen der Benutzeroberfläche an. Diese Komponenten agieren mit dem Nutzer. Ein Beispiel für eine Komponente ist ein `Button`. Genau wie in *Swing* stellt diese Komponente eine grafische Schaltfläche dar, die der Nutzer anklicken kann. Zusätzlich zu den bereits vorhandenen können auch eigene Komponenten entwickelt werden. Außerdem existieren schon viele Add-ons die *Vaadin* um weitere grafische Komponenten erweitern. Mit Hilfe von Layouts werden die Komponenten auf der Benutzeroberfläche plaziert. Ein Layout ist für die Größe und die Position einer oder mehrerer Komponenten zuständig. Um auf Benutzereingaben zu reagieren, müssen die Komponenten mit einem „listener“ verbunden werden. Dieser „listener“ wird dann bei jeder Benutzereingabe benachrichtigt. An einen `Button` kann zum Beispiel ein `ClickListener` gebunden werden. Dieser `ClickListener` wird dann bei einem Mausklick auf den entsprechenden `Button` aufgerufen.

[4] *API* (Application Programming Interface) steht für die Programmierschnittstelle.
[5] *Swing* ist eine *Java*-Standard-Bibliothek zum Programmieren von grafischen Oberflächen.

Mit *Vaadin* können sowohl clientseitige, als auch serverseitige Applikationen programmiert werden.
Clientseitige Applikationen werden in *Java* geschrieben und durch den *Vaadin* Compiler in *JavaScript* übersetzt. Die Applikation läuft dann im Browser.
Die serverseitigen Applikationen sind umfangreicher und besser unterstützt. Ein sogenanntes *Vaadin Servlet* läuft hierbei in einem *Java* Applikationsserver und beantwortet *HTTP* Anfragen. Jede Interaktion des Clients mit einer grafischen Komponente wird als ein Event interpretiert. Das *Vaadin Servlet* arbeitet auf der *Java Servlet API*. Es erhält die Events von verschiedenen Clients und entscheidet, zu welcher Sitzung diese Anfrage gehört. Die Sitzung des Nutzers wird hierbei über Cookies verfolgt. Das *Servlet* delegiert die Anfragen an die entsprechende Sitzung. Dort wird entschieden, zu welcher Komponente das Event gehört. Der „listener“, welcher mit der Komponente verbunden ist, bekommt dann das Event.

3.5.2 Architektur von *Vaadin*

Das Kernstück von *Vaadin* ist das „User Interface“, es stellt die grafische Oberfläche dar. Es kreiert die Nutzeroberfläche aus den im vorherigen Abschnitt beschriebenen grafischen Komponenten. Außerdem richtet es die „listener“ ein, um die Nutzereingabe zu behandeln. Das „User Interface“ kann im Browser über eine URL erreicht, oder in eine *HTML* Seite integriert werden.

Das Aussehen der grafischen Komponenten wird nicht in *Java* sondern in *CSS* (Cascading Style Sheets) oder *Sass* (Syntactically Awesome Style Sheets) definiert. *CSS* und *Sass* sind Sprachen, welche Darstellungsvorgaben für *HTML* definieren. *Vaadin* bietet viele schon vordefinierte sogenannte „themes“, die das Aussehen der Komponenten bestimmen.

Wie bereits beschrieben, wird die Nutzereingabe in Events interpretiert und an das *Servlet* weitergeleitet. Zusätzlich hierzu kann durch einen „server push“ das „User Interface“ vom Server aus verändert werden. Bei einem „server push“ werden die Änderungen direkt vom Server zum Client gesendet, ohne dass der Client eine Anfrage senden muss. Dadurch kann die Nutzeroberfläche direkt vom Server aus aktualisiert werden.

Das Back-End einer Software kann unabhängig von *Vaadin* individuell gestaltet werden. Das hat den Vorteil, dass die Software dynamisch bleibt und bei Änderungen der Oberfläche nicht die gesamte Software geändert werden muss. Es werden jedoch standardisierte Schnittstellen geboten, um Komponenten direkt an das Datenmodell des Back-End zu knüpfen. Dadurch haben die Komponenten direkten Zugriff auf die Daten und können diese verändern.

3.6 Projektverwaltung mit *Maven*

Maven ist ein von der Apache Software Foundation entwickeltes Tool zur Verwaltung eines Softwareprojektes. Es kann das Projekt sowie die Dokumentation bauen und übernimmt die Berichterstattung.

Maven ist in *Java* entwickelt und dient insbesondere der Verwaltung von *Java* Projekten. Es standardisiert den Aufbau von Projekten. Abhängigkeiten werden automatisch aufgelöst, aus *Maven*-Repositories heruntergeladen und aktualisiert. *Maven* lässt automatisch *Unit* Tests laufen und erstellt Testberichte.

Konfiguriert wird *Maven* in *XML* über eine Konfigurationsdatei namens „pom.xml“ (Project Object Model). Mit *Maven* lassen sich einfach Projekte einrichten, und der Entwickler wird zu „best practice“ Richtlinien geleitet. Je standardisierter das Projekt aufgebaut wird, desto weniger muss *Maven* konfiguriert werden. *Maven* gibt einen standardisierten Lebenszyklus vor, in dem die Software erstellt werden soll. Dieser umfasst unter anderem das Kompilieren, Testen und Installieren.

Maven ist durch eigene in *Java* geschriebene Funktionen erweiterbar. Es kann „.jar“ Dateien für Applikationen bauen, ist aber auch für die Webentwicklung geeignet und kann „.war“ Archive packen.

3.7 Nutzerverwaltung mittels *LDAP*

LDAP (**L**ightweight **D**irectory **A**ccess **P**rotocol) ist ein Anwendungsprotokoll zum Erfragen und Modifizieren von Informationen aus einer verteilten hierarchischen Datenbank [2].

LDAP arbeitet nach dem Client-Server Modell. Der *LDAP*-Client kann z.B. ein Browser, ein E-Mail Client oder eine Webapplikation sein. Die Daten sind auf dem *LDAP*-Server in einem *LDAP*-Verzeichnis abgespeichert.

Ein *LDAP*-Verzeichnis hat eine hierarchische Baumstruktur, vergleichbar mit *DNS*[6]. Die Wurzel des Baumes stellt das oberste Datenobjekt dar. Alle weiteren Daten werden als Äste und Blätter untergeordnet. Diese Verzeichnisstruktur ist optimiert für Lesezugriffe und somit geeignet für Daten, die oft gelesen, aber selten geschrieben werden. Die hierarchische Struktur kann effizient durchsucht werden, da bei der Suche nach einzelnen Daten nur die jeweils übergeordnete Gruppe durchsucht werden muss. Hierbei werden alle übrigen Gruppen der gleichen hierarchischen Ebene ignoriert. Zudem erlaubt die Baumstruktur, die Daten über Netze zu verteilen. Die Daten können leicht repliziert werden. Obwohl das *LDAP*-Verzeichnis alle möglichen Daten enthalten kann, ist es besonders gut geeignet zum Speichern von Nutzerdaten. Einzelne Nutzer werden in Gruppen zusammengefasst,

[6] Das **D**omain **N**ame **S**ystem (DNS) übersetzt eine menschenlesbare URL in die zugehörige IP-Adresse.

die hierdurch sich ergebenden Rechte müssen öfter gelesen als geschrieben werden, zudem müssen sie immer schnell bereit gestellt werden.

LDAP ist ein asynchrones Protokoll. Das heißt wenn ein Client gleichzeitig mehrere Anfragen an den Server sendet, ist unbestimmt welche Antwort zuerst zurückkommt.

LDAP wird auf den HPC-Systemen des *JSC* für die Authentisierung von Nutzern verwendet.

3.8 Authentifizierung mittels *SAML*

SAML (**S**ecurity **A**ssertion **M**arkup **L**anguage) ist ein *XML* basiertes Protokoll und Framework zum Austausch von Sicherheitsinformationen zwischen vernetzten Domänen. *SAML* wurde von dem Security Services Technical Committee (SSTC) der Organisation OASIS (**O**rganization for the **A**dvancement of **S**tructured **I**nformation **S**tandards) entwickelt [11].

SAML wird z.B. verwendet um Authentifizierungsinformationen zwischen einem Identity Provider und einem Service Provider auszutauschen. Ein Identity Provider identifiziert bzw. authentifiziert Nutzer und bestätigt die Authentifizierung gegenüber dem Service Provider. Der Service Provider stellt dem Nutzer verschiedene Dienste zur Verfügung und kümmert sich um die Autorisierung des Nutzers.

SAML definiert keine Regeln, wie der Nutzer authentifiziert oder autorisiert wird, sondern nur wie diese Informationen ausgetauscht werden.

Eine der Aufgaben von *SAML* ist es eine „Single Sign-On“ Funktion bereitzustellen. „Single Sign-On“ bedeutet, dass ein Nutzer nach der erfolgreichen Anmeldung an einer Anwendung automatisch auch zur Benutzung weiteren Anwendungen berechtigt ist und bei diesen automatisch mit gleichen Nutzerinformationen angemeldet wird. Die Nutzerdaten können hierbei über Cookies gespeichert werden. Der Nachteil an Cookies ist jedoch, dass diese nicht zwischen *DNS* Domänen ausgetauscht werden. Dadurch können die Nutzerdaten nicht über Domänen hinweg verwendet werden. Um trotzdem ein „Single Sign-On“ über Domänengrenzen hinweg zu gewährleisten kann *SAML* genutzt werden. *SAML* bietet ein standardisiertes Protokoll zum Übertragen der Nutzerdaten zwischen unabhängigen Webservern. So können selbst in heterogenen Umgebungen Nutzerdaten ausgetauscht werden.

Eine weitere Aufgabe von *SAML* ist es eine föderierte Identität zu erstellen. Damit ist eine gemeinschaftliche und geteilte Identifikation eines Nutzers über Domänengrenzen hinweg gemeint. *SAML* definiert nicht nur, wie die Nutzerdaten ausgetauscht werden, sondern bietet außerdem ein einheitliches Abbild des Nutzers, dessen Nutzerdaten ausgetauscht wurden. So können Informationen über den Nutzer in verschiedenen Webdiensten ausgetauscht werden. Nutzer haben oft individuelle, lokale Nutzeridentitäten in den verschiedenen Domänen. Durch die föderierte Identität teilen alle Domänen jedoch die selben Informationen. Durch den Einsatz von *SAML* werden Nutzerverwaltungskosten gespart, da nun nicht mehr

jeder Dienst einzeln Daten sammelt.

Die Sicherheitsinformationen werden in „assertions“ (Behauptungen, Thesen) dargestellt, welchen die Applikationen über Sicherheitsgrenzen weg vertrauen können. Die „assertions“ werden in *XML* dargestellt und durch eine „assertion ID“ identifiziert. Jede „assertion“ speichert einen Fakt über ein Subjekt z.B. eine Eigenschaft eines Nutzers. Das Subjekt wird hierbei durch einen Namen und eine Sicherheitsdomäne identifiziert. In jeder „assertion“ steht der Aussteller und der Zeitpunkt der Ausfertigung. *SAML* definiert präzise Regeln zum Erfragen, Erstellen, Kommunizieren und zur Benutzung von „assertions“. Es gibt drei verschiedene Arten von „assertions“:

- Authentifizierung
  Gibt an, dass das Subjekt authentifiziert wurde. Zusätzlich wird noch das Kriterium festgehalten, auf welche Art und Weise und wann die Authentifizierung stattfand.

- Autorisierung
  Gibt an, dass das Subjekt autorisiert wurde auf eine bestimmte Ressource zuzugreifen. Die Ressource ist z.B. eine Webseite.

- Attribut
  Weist dem Subjekt verschiedene Attribute zu. Ein Attribut ist z.B. die Nutzergruppe eines Nutzers.

3.9 Identitäts-Management mit *Unity*

Unity (**UNI**fied identi**TY**) ist eine Software zum Identitäts-Management [15]. Sie ist in *Java* implementiert und OpenSource. Die Software bietet einen sicheren Weg um die Authentisierung für Web und Cloud Ressourcen zu übernehmen. *Unity* dient unter anderem der Verwaltung von föderierten Identitäten (siehe Kapitel 3.8).

Unity authentifiziert Personen oder Server auf verschiedene Arten, wie z.B. anhand von Passwort und Nutzername oder über Zertifikate. Die Verifizierung stellt *Unity* dann einem Webdienst mittels eines Protokolls wie SAML zur Verfügung. Hierbei kann *Unity* die Authentifizierung selbst übernehmen. Die Nutzerdaten sind dann in einer lokalen Datenbank registriert. Die Software kann den Nutzer aber auch an Drittanbieter weiterleiten, wo er dann authentifiziert wird. So kann sie z.B. als Brücke zu Benutzerdatenbanken dienen, welche aktuelle Webstandards nicht unterstützen.

Weiterhin kann *Unity* auch die Verwaltung von Attributen, Rechten und Gruppen übernehmen. *Unity* ist leicht erweiterbar und es existieren schon viele Erweiterungen, wie z.B. verschiedene Methoden zur Authentifizierung.

Durch eine aufgeräumte grafische Oberfläche ist *Unity* sehr intuitiv, einfach zu bedienen und somit benutzerfreundlich.

4 Programmkonzeption

Die Applikation wurde nach dem *MVC*-Muster (**M**odel **V**iew **C**ontrol) entworfen. Das *MVC*-Muster ist ein Entwurfsmuster zur Strukturierung eines Softwareentwurfs [7]. Ziel des Musters ist eine möglichst flexible Softwarearchitektur. Die Software wird in drei Schichten eingeteilt: das Datenmodell (Model), die Präsentationsschicht (View) und die Logik (Control). Die genauen Aufgaben und die Konzeption der einzelnen Schichten werden im Folgenden beschrieben.

4.1 Logik der Webanwendung

Die Logik ist für die gesamte Programmsteuerung zuständig. Sie empfängt die Eingabe des Nutzers von der Präsentationsschicht und interpretiert sie. Anschließend lässt die Logik die Präsentationsschicht und das Datenmodell entsprechend reagieren. Die Logik ist im Konzept komplett von der Benutzeroberfläche getrennt. Damit ist sichergestellt, dass in weiteren Schritten weitere Benutzerschnittstellen implementiert werden können, ohne dass die Logik geändert werden muss. Die Schnittstellen der Logik müssen hierfür klar definiert sein und bestmögliche Modularität gewährleisten.

4.1.1 Nutzerverwaltung

Die Nutzer dieser Applikation werden über föderierte Identitäten verwaltet. Diese Identitäten stellt die *DFN-AAI* (**D**eutsches **F**orschungs**N**etz **A**uthentifikations- und **A**utorisierungs-**I**nfrastruktur) zur Verfügung [5].

Im Deutschen Forschungsnetz (*DFN*) laufen verschiedene Dienste, z.B. die *DFN-AAI*. Das *DFN* ist ein wissenschaftliches Netz in Deutschland, welches Hochschulen und Forschungseinrichtungen verbindet. Es interagiert aber auch mit anderen Forschungsnetzen weltweit und ist mit dem Internet verbunden. Die *DFN-AAI* Föderation bietet eine Infrastruktur zum Austausch von Authentifizierungs- und Autorisierungsinformationen zwischen wissenschaftlichen Einrichtungen aber auch kommerziellen und nicht kommerziellen Anbietern. Der Zugang zu den Informationen soll vereinheitlicht und vereinfacht werden. Die Föderation schafft ein Vertrauensverhältnis. So sind die Föderationsmiglieder vertraglich an Abmachungen gebunden, die sie in Bezug auf „Identity-Management“ einhalten müssen.

Die Webapplikation gewährt allen Mitgliedern der *DFN-AAI* Zugriff. Jeder Nutzer wird

durch seine E-Mail-Adresse eindeutig identifiziert. Die Authentifizierung des Nutzers geschieht hierbei über einen *Unity*-Webserver (siehe Kapitel 4.4.3).

4.1.2 Integration ins *UFTP*

Die Webapplikation übernimmt im *UFTP* Modell die Aufgaben des Authentifizierungsservers und des *UFTP*-Clients (siehe Kapitel 3.3.2). Der Nutzer authentisiert sich gegenüber der Applikation mit Hilfe von *Unity*, und der Webserver initiiert dann eine *UFTP*-Sitzung mit dem *UFTP*-Server. Anschließend fungiert die Applikation als Vermittler zwischen dem Browser des Nutzers und dem *UFTP*-Server.

Diese Architektur hat den Vorteil, dass der Webserver nicht die Daten der Dateisysteme besitzen muss. Der Webserver kann auf einem eigenen Rechner laufen, der nicht zu *JUST* gehören muss, er benötigt nur Zugriff auf den *UFTP*-Server.

4.1.3 Konzept einer Nutzersitzung

Die Sitzung eines Nutzers startet, sobald er die erste Anfrage an die Webapplikation stellt. Durch die Authentisierung verschafft er sich Zugriff auf seine Dateien. Nicht alle Nutzer des *DFN-AAI* haben auch ein *LDAP*-Benutzerkonto auf *JUST*. Nach der Authentifizierung gleicht die Webapplikation die E-Mail-Adresse des Nutzers gegenüber dem Eintrag am *LDAP*-Server des *JSC* ab. Falls der Nutzer im *LDAP*-Server gespeichert ist, besitzt er auch ein Benutzerkonto auf *JUST* und somit ein Heimatverzeichnis. Alle anderen Nutzer des *DFN-AAI* ohne Heimatverzeichnis können nur auf die ihnen freigegebenen Dateien bzw. Verzeichnisse zugreifen.

Nach der Authentifizierung gibt es zwei Instanzen, die für die Verwaltung der Sitzung zuständig sind. Dies sind der `FileManager` und der `ACLManager` (*ACL* steht für Access Control List). Für jeden angemeldeten Nutzer werden ein neuer `FileManager` sowie ein neuer `ACLManager` erstellt.

Der `FileManager` verwaltet alle Dateien die der Nutzer selbst besitzt. Er stellt Methoden zur Verfügung, um sich die Dateien eines Nutzers auflisten zu lassen und um neue Dateien hochladen zu können. Nach der Authentifizierung des Nutzers initiiert der `FileManager` eine *UFTP* Nutzersitzung beim *UFTP*-Server unter dem Konto des Nutzers. Der `FileManager` verwaltet diese Sitzung und schließt sie, sobald der Nutzer sich abmeldet.

Der `ACLManager` verwaltet alle Freigaben, die den Nutzer betreffen. Einerseits ist er dafür zuständig, Freigaben zu speichern, die der Nutzer auf seine Daten setzt. Andererseits kennt er alle Daten, die dem Nutzer freigegeben wurden. Wenn ein Nutzer auf die freigegebenen Daten eines anderen Nutzers zugreift, geschieht dies über neue *UFTP* Datenverbindungen. Hierbei meldet sich der `ACLManager` beim *UFTP*-Server mit den Benutzerdaten des eigentlichen Besitzers der Dateien an (Näheres in Kapitel 4.1.4). Die Dateien werden hierbei nach Nutzern gruppiert ausgelesen. Im Gegensatz zum Zugriff auf eigene Dateien, wird die Da-

tenverbindung nach dem Zugriff auf freigegebene Dateien jedoch wieder beendet. Dies hat den Vorteil, dass der *UFTP*-Server weniger offene Datenverbindungen mit dem Webserver besitzt. Sonst müsste für alle freigegebenen Dateien jeweils eines anderen Nutzers eine neue *UTFP*-Sitzung geöffnet werden.

4.1.4 Freigabe von Daten

Jeder Nutzer kann seine Dateien beliebig freigeben. Das heißt er kann einem oder auch mehreren anderen Nutzern Leserechte und optional auch Schreibrechte auf seine Dateien geben.

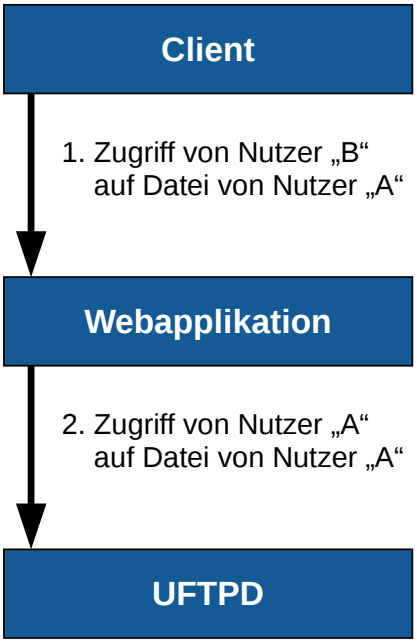

Abbildung 4.1: Zugriff auf eine freigegebene Datei

Die Freigaben werden vom Webserver gespeichert und werden den Dateisystemen nicht mitgeteilt. Wenn ein Nutzer „A“ einem anderen Nutzer „B“ eine Datei freigibt, merkt sich dies der Webserver. Wenn nun Nutzer „B“ auf die Datei von Nutzer „A“ zugreifen möchte, überprüft der Webserver zunächst in seiner eigenen Datenbank, ob der Zugriff berechtigt ist. Falls der Zugriff erlaubt ist, meldet sich der Webserver als Nutzer „A“ beim Dateisystem an und führt den von Nutzer „B“ gewünschten Dateizugriff durch. Der Nutzer bekommt von diesem Vorgehen nichts mit. Für ihn funktioniert der Dateizugriff auf freigegebene Daten genau wie der Zugriff auf die eigenen Daten.

Eine Freigabe kann sich auf eine Datei beziehen, aber auch auf ein ganzes Verzeichnis. Die Freigabe auf ein Verzeichnis gilt immer für alle Dateien in dem freigegebenen Verzeichnis und wirkt rekursiv auf alle Unterverzeichnisse.

Ein Nutzer sieht jederzeit eine Liste aller Dateien bzw. Verzeichnisse, die ihm freigegeben wurden. Da die Daten auf den Dateisystemen und nicht auf dem Webserver liegen, kann es sein, dass Daten gelöscht werden, ohne dass der Webserver dies mitbekommt. Da die Freigaben jedoch auf dem Webserver gespeichert sind, kann es nun passieren, dass eine Datei gelöscht wurde, die zugehörige Freigabe aber noch existiert. Um zu verhindern, dass der Nutzer solche fehlerhaften Freigaben sieht, muss der Webserver in regelmäßigen Intervallen und bei jedem Dateizugriff überprüfen, ob die Datei noch existiert.

Der Besitzer freigegebener Daten kann die Freigaben jederzeit wieder aufheben. Auch ein Nutzer, der eine Datei freigegeben bekommen hat, kann diese Freigabe wieder aufheben, damit er die Daten nicht länger angezeigt bekommt. Er kann jedoch nur die auf ihn bezogenen Freigaben löschen, nicht aber Freigaben für andere Nutzer. Außerdem kann ein Nutzer eine Datei, die ihm freigegeben wurde, nicht weiteren Nutzern freigeben.

In Kapitel 4.2.1 werden alle Informationen, die in einer Freigabe gespeichert werden, genau aufgelistet.

4.1.5 Automatische Aktualisierung

Die Daten auf den Dateisystemen können auch von außerhalb der Applikation verändert werden. Der Nutzer kann sich zum Beispiel direkt auf ein Dateisystem anmelden und dort Dateien oder Verzeichnisse löschen, umbenennen und vieles mehr. Die Applikation wird jedoch nicht über die Änderungen der Daten im Dateisystem informiert. Es kann also passieren, dass angezeigte Daten gar nicht mehr existieren oder verändert wurden. Damit der Nutzer jedoch trotzdem die aktuellsten Daten angezeigt bekommt, muss die Applikation alle Informationen regelmäßig aktualisieren und mit den Dateisystemen synchronisieren.

Es kann auch passieren, das freigegebene Dateien oder Verzeichnisse von anderen Nutzern verändert wurden. Zum Beispiel könnte es passieren, dass Rechte bzw. Freigaben sich während einer Sitzung verändern. Dies ist der Fall, wenn der Besitzer der Datei während der Sitzung die Freigabe geändert oder gelöscht hat. In diesem Fall darf die Datei natürlich nicht geöffnet und im besten Fall noch nicht einmal angezeigt werden.

Somit muss vor jedem Zugriff auf eine Datei oder ein Verzeichnis überprüft werden, ob die Datei bzw. das Verzeichnis noch auf dem Dateisystem existiert. Falls es sich um eine freigegebene Datei oder ein freigegebenes Verzeichnis handelt, muss zusätzlich jedes mal überprüft werden, ob der Zugriff auch berechtigt ist, oder ob der Besitzer der Datei die Freigabe eventuell zwischenzeitlich aufgehoben hat.

Um oben genannte Probleme zu beheben aktualisiert die Applikation ihre Informationen selbstständig in regelmäßigen Abständen.

4.1.6 Logische Architektur

Zusammengefasst ergibt sich folgende logische Architektur:

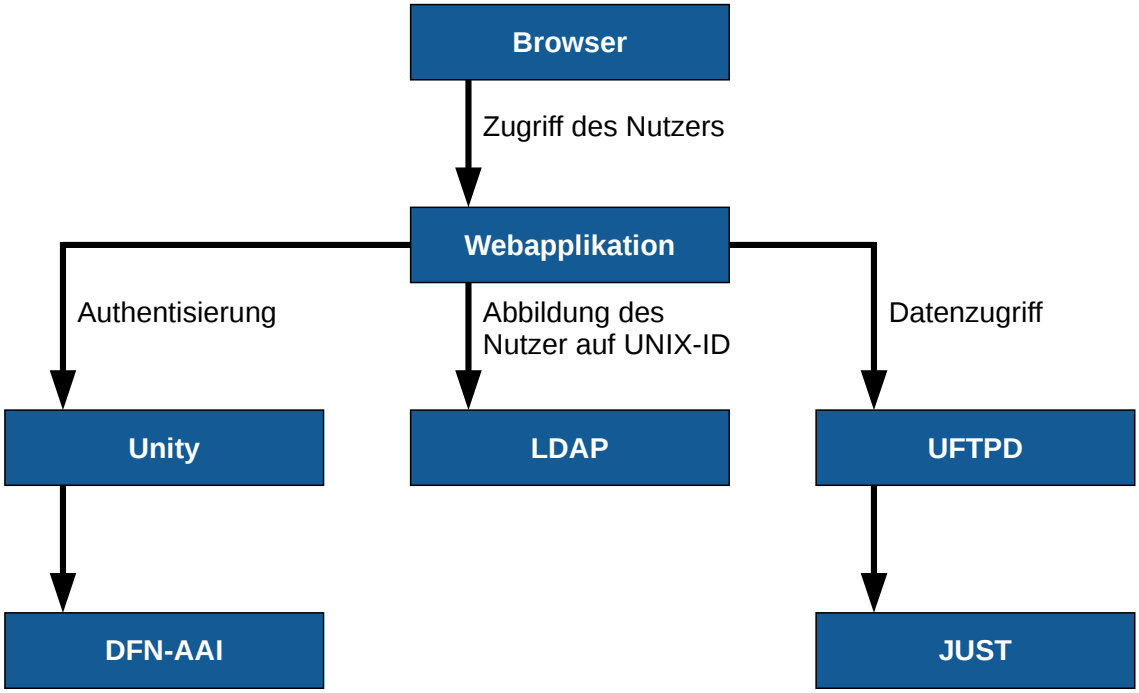

Abbildung 4.2: Logische Architektur der Webanwendung

Der Nutzer greift über seinen Browser auf die Webapplikation zu.

Zunächst authentisiert er sich über *Unity* als ein Mitglied des *DFN-AAI* und wird anschließend durch seine E-Mail-Adresse eindeutig identifiziert.

In einem weiteren Schritt muss die Applikation nun den Nutzer auf die entsprechende *UNIX-ID* abbilden. Hierzu gleicht sie die E-Mail-Adresse des Nutzers gegenüber dem Eintrag am *LDAP*-Server des *JSC* ab.

Für den Datenzugriff startet die Webapplikation unter der entsprechenden User-ID des Besitzers eine neue *UFTP*-Sitzung beim *UFTPD*. Dieser greift dann direkt auf *JUST* zu. Falls der Nutzer eine *UNIX-ID* auf *JUST* besitzt, bekommt er sein Heimatverzeichnis und alle seine Dateien angezeigt. Zusätzlich bekommen alle Nutzer die ihnen freigegebenen Dateien angezeigt.

4.2 Datenmodell der Webanwendung

Das Datenmodell stellt die Datenhaltung dar und enthält die gesamte Zustands- und Datenlogik. Es ist völlig autark, somit weiß es nichts über die Präsentationsschicht bzw. die Logik. Es stellt Schnittstellen bereit, über die der Zustand der Daten abgefragt und beeinflusst werden kann. Das Datenmodell kann die Präsentationsschicht über Zustandsänderungen informieren.

4.2.1 Aufbau einer Freigabe

Eine Freigabe besteht aus folgenden Informationen:

- Welche Datei oder Verzeichnis?
  Zusätzlich zum Namen der Datei wird der absolute Pfad, das Datum und die Größe der Datei gespeichert. Näheres in Kapitel 4.2.2
- Wem gehört die Datei bzw. das Verzeichnis?
  Zu jeder Freigabe wird auch der Besitzer der Datei registriert.
- Wer bekommt die Freigabe?
  Daten können einem Nutzer freigegeben werden. Die Freigabe kann aber auch für Gruppen oder Rollen[1] gelten. Dies macht es möglich die Dateien auch dem System unbekannten Nutzern freizugeben. Mehr darüber in Kapitel 4.2.3
- Welche Rechte?
  Eine Freigabe bedeutet immer, dass der Nutzer die Daten lesen darf. Optional kann ein Nutzer anderen Nutzern jedoch auch Schreibrechte auf seine Daten einräumen. Eine Freigabe auf ein Verzeichnis wirkt sich immer rekursiv auf alle Unterverzeichnisse und Dateien aus.

4.2.2 Interface `IPath`

Ein `IPath` stellt eine einzelne Datei oder ein einzelnes Verzeichnis dar. Es ist eine Schnittstelle um auf ein beliebiges Dateisystem zuzugreifen. Eine konkrete Implementierung des Interface könnte zum Beispiel auf dem `java.io.File` basieren und somit auf dem lokalen Dateisystem arbeiten. Für diese Implementierung wird ein `IPath` basierend auf *UFTP* implementiert.

Die Methoden, die ein `IPath` bereitstellt, sind in Abbildung 4.3 dargestellt.

[1] Eine Rolle vergibt Rechte für alle ihr zugeordneten Nutzer, dadurch werden mehreren Nutzern die gleichen Rechte zugeordnet. Es werden immer nur Rollen verteilt und keine einzelnen Rechte für Nutzer mehr vergeben.

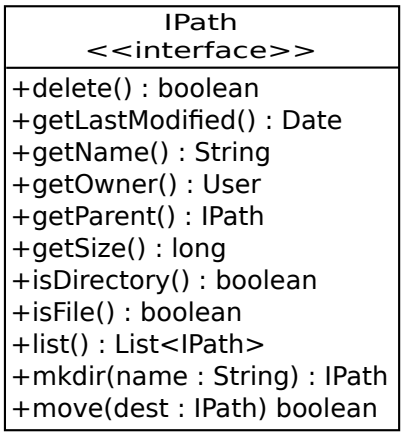

Abbildung 4.3: Interface `IPath`

- `delete() : boolean`
  Es wird das Verzeichnis bzw. die Datei gelöscht. Bei Erfolg wird `True` zurückgegeben.
- `getLastModified() : Date`
  Gibt das Datum der letzten Änderung der Datei bzw. des Verzeichnisses zurück.
- `getName() : String`
  Diese Methode gibt den Namen der Datei bzw. des Verzeichnisses zurück.
- `getOwner() : User`
  Als Ergebnis liefert diese Methode den Besitzer der Datei bzw. des Verzeichnisses.
- `getParent() : IPath`
  Gibt das Verzeichnis zurück, in dem die Datei bzw. das Verzeichnis liegt.
- `getSize() : long`
  Diese Methode gibt die Größe der Datei bzw. des Verzeichnisses zurück.
- `isDirectory() : boolean`
  Es wird überprüft, ob es sich um ein Verzeichnis handelt.
- `isFile() : boolean`
  Prüft, ob es sich um eine Datei handelt.
- `list() : List<IPath>`
  Gibt alle `IPath` zurück, die in diesem Verzeichnis liegen.
- `mkdir(name : String) : IPath`
  Erstellt ein neues Verzeichnis in dem aktuellen Verzeichnis.
- `move(dest : IPath) : boolean`
  Verschiebt die Datei bzw. das Unterverzeichnis in das neue Verzeichnis namens `dest`.

4.2.3 Interface `Target`

Ein `Target` ist das Ziel einer Freigabe. Ein `Target` kann für einen einzelnen Nutzer stehen, kann aber auch eine Gruppe oder eine Rolle repräsentieren. Im Programm wird ein `Target` als ein Interface dargestellt, siehe Abbildung 4.4.

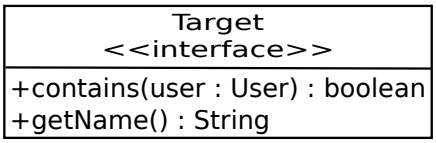

Abbildung 4.4: Interface `Target`

Das Interface hat zwei Methoden.

- `contains(user : User) : boolean`
  Bekommt einen `user` übergeben und gibt `True` zurück, falls der `user` zu dieser Gruppe oder Rolle gehört. Falls das `Target` einen einzelnen Nutzer darstellt, gibt die Methode `True` zurück, wenn der übergebene `user` dem Nutzer entspricht.

- `getName() : String`
  Die Methode gibt einen eindeutigen Namen für das `Target` zurück.

4.2.4 Klasse `User`

Die Klasse `User` identifiziert einen einzelnen Nutzer. Sie implementiert das Interface `Target`. Ein Nutzer wird durch seine föderierte Identität gekennzeichnet (siehe Kapitel 4.1.1). Er erhält einen eindeutigen Nutzernamen (seine E-Mail-Adresse) zur Identifikation, zudem können ihm an dieser Stelle bzgl. der Erweiterbarkeit weitere Eigenschaften zugewiesen werden.

4.3 Präsentationsschicht der Webanwendung

Die Präsentationsschicht ist für die Darstellung zuständig und stellt die Benutzeroberfläche dar. Sie erhält die Daten aus dem Datenmodell und präsentiert sie. Der Benutzer interagiert mit der Präsentationsschicht. Jede Benutzereingabe wird an die Logik weitergereicht und dort verarbeitet.

4.3.1 Konzept der Benutzeroberfläche

Als Schnittstelle für den Nutzer dient eine Webapplikation. Diese ist von jeder Plattform aus leicht zu erreichen, wobei das Betriebssystem keine Rolle spielt. Allein eine Internet- bzw. Intranetanbindung und ein Browser werden für die Nutzung der Applikation benötigt. Eine Netzanbindung ist jedoch ohnehin erforderlich, um die Dateien von entfernten Systemen abrufen zu können.

Zusätzlich hat eine Webapplikation den Vorteil, dass sie leicht zu warten ist. Aktualisierungen werden zentral auf dem Server vorgenommen und müssen nicht beim Client aufgespielt werden. Durch dieses Verfahren wird der Nutzer bei einer Aktualisierung oder Fehlerkorrekturen nicht in seiner alltäglichen Arbeit durch manuelle oder automatische Installationsroutinen beeinträchtigt.

Beim Starten der Applikation wird der Nutzer durch *Unity*, beispielsweise durch eine Anmeldung mit Nutzernamen und Passwort, authentifiziert. Jedoch wird die Schnittstelle zu *Unity* nicht im Rahmen dieses Projektes entwickelt. Als Alternative bekommt jeder nicht angemeldete Besucher der Seite vorerst eine Startseite mit zwei Eingabefeldern angezeigt, in die er seine Kenndaten einträgt. Diese Daten werden zur Zeit mit einer lokalen Nutzerdatenbank abgeglichen.

Die Benutzeroberfläche besteht anschließend aus einer einzigen Seite. So bekommt der Nutzer jederzeit alle Informationen auf einen Blick angezeigt. Zusatzinformationen können durch Popups eingeblendet werden. Dadurch gewinnt die Oberfläche stark an Übersichtlichkeit und dem Nutzer bleiben unnötige Klicks erspart. *Vaadin* ist für einseitige Webapplikationen gedacht und eignet sich daher sehr gut für dieses Projekt. Die Dateien des Nutzers werden in einem Dateibaum angezeigt, da diese Art der Darstellung intuitiv und den meisten Nutzern schon bekannt ist. Es wird zu jeder Datei die Größe sowie das Datum der letzten Änderung so angezeigt, dass sie der entsprechenden Datei zugeordnet werden können. Des weiteren werden im Dateibaum, neben den Dateien des Nutzers, auch die Dateien angezeigt, die dem Nutzer freigegeben wurden. Alle freigegebenen Dateien werden in einem eigenen Bereich aufgelistet. Zudem wird zu jeder Datei ihr Besitzer dargestellt, außerdem, ob und für wen sie freigegeben wurde. Für den Spezialfall, dass Nutzer kein eigenes Heimatverzeichnis auf dem Dateisystem besitzen, bekommen diese nur die ihnen freigegebenen Dateien und Verzeichnisse angezeigt.

Über Schaltflächen in der Oberfläche werden Funktionen zur Datenverwaltung bereitge-

stellt. Hierzu gehören Schaltflächen zum Dateiaustausch sowie Funktionen zum Umbenennen, Löschen oder Erstellen von Dateien.

4.3.2 Container `IContainer`

Die Anbindung der Daten an die grafische Benutzeroberfläche geschieht über einen Container namens `IContainer`. Da dieser auf den eigens entwickelten Schnittstellen zum Datenmodell basieren soll und in *Vaadin* hierzu keine Herangehensweise zur Verfügung stand, musste dieser komplett selbst entwickelt werden. Der Container repräsentiert Daten, welche in Form einer Tabelle dargestellt werden. Er besteht wiederum aus mehreren Elementen, welche verschiedene Eigenschaften aufweisen können. Die Elemente werden in der Tabelle zeilenweise dargestellt, deren Spalten verschiedene Eigenschaften darstellen.

Die vom Container `IContainer` repräsentierten Daten sind die Dateien und Verzeichnisse des Nutzers. Diese Informationen erhält der Container über den `FileManager` und den `ACLManager` (siehe Kapitel 4.1.3). Der `IContainer` lässt sich vom `FileManager` zunächst alle Dateien des Nutzers, und zusätzlich vom `ACLManager` alle freigegebenen Dateien auflisten.

Jedes Element des Containers ist ein `IPath` (siehe Kapitel 4.2.2). Ein Element hat die Eigenschaften „name“, „last modified“, „size“ und „owner“. Diese Eigenschaften erhält der `IContainer` direkt über den `IPath`, indem es die entsprechenden Methoden aufruft. Weiterhin hat jedes Element noch die Eigenschaft „shared with“. Diese gibt an, ob die Datei bzw. das Verzeichnis freigegeben wurde und für wen. Diese Information erhält der Container vom `ACLManager`.

Der `IContainer` hat zusätzlich zu den Grundfunktionen eines Containers noch weitere Funktionen. Zum einen sind die Elemente hierarchisch sortiert. Angefangen mit dem Heimatverzeichnis des Nutzers sind alle Unterverzeichnisse in einer Baumstruktur angeordnet. Außerdem können die Elemente nach Namen, Änderungsdatum oder Größe aufsteigend bzw. absteigend sortiert werden.

4.4 Sicherheit der Webanwendung

Die Sicherheit der Applikation ist eines der Hauptanforderungskriterien des Projektes. Die verschiedenen Sicherheitsmaßnahmen werden im folgenden detailliert beschrieben.

4.4.1 Verschlüsselter Kanal durch *TLS*

Die Webanwendung wird zur elementaren Absicherung über *TLS* gesichert. Die Kommunikation läuft über einen verschlüsselten Kanal, so ist die Vertraulichkeit und Integrität der Verbindung gewährleistet.

Der Server wird zertifiziert und kann sich so eindeutig gegenüber dem Browser authentisieren. Dadurch können die Nutzer sicher stellen, dass sie wirklich mit dem Server kommunizieren, den sie erwarten.

Durch die Verschlüsselung der Kommunikation ist eine private Verbindung gewährleistet. Daten können nicht von Dritten mitgelesen werden

4.4.2 Sicherheit durch *UFTP*

UFTP ist ein sicheres Transferprotokoll, über welches auf die Daten der Nutzer von *JUST* zugegriffen werden kann (siehe Kapitel 3.3.6). Es eignet sich hervorragend für dieses Projekt und bietet eine vertrauenswürdige Methode um auf die Daten der Dateisysteme zuzugreifen.

In HPC-Systemen ist der Zugriff auf die Daten meist ein sicherheitskritischer Faktor. Zum Beispiel ist es sehr problematisch, wenn auf die Daten als `root` zugegriffen werden muss. Der Zugriff mit allumfassenden Berechtigungen stellt ein Sicherheitsproblem dar. Bei *UFTP* dagegen geschieht der Datenzugriff jedoch stets unter der User-ID des eigentlichen Nutzers. Somit kann jeder Nutzer nur seine eigenen Dateien verändern und der Schaden ist im schlimmsten Fall begrenzt auf das eigene Verzeichnis. Weiterhin benötigt *UFTP* kein eigenes Benutzerkonto auf *JUST*, unter welchem die Dateien abgespeichert werden müssen.

Die Struktur der Dateisysteme muss nicht geändert werden, denn die Daten bleiben auf den Dateisystemen liegen. Dadurch müssen keine Sicherheitsrichtlinien geändert werden und die Daten werden nicht in vertrauensunwürdige Umgebungen kopiert bzw. verschoben.

Weiterhin bietet *UFTP* die Möglichkeit, durch den separaten Authentifizierungsserver, eine eigene sichere Authentisierung zu implementieren (siehe Kapitel 4.4.3).

Da *UFTP* eine Eigenentwicklung des *JSC* ist und auch bereits im Einsatz erprobt wurde, ist der Umgang hiermit bekannt. Zudem ist es sehr gut für große Datenmengen geeignet, wie sie im HPC-Bereich anfallen.

4.4.3 Sichere Authentisierung

Die Nutzerdaten der internen Mitarbeiter des *FZJ* für die HPC-Systeme werden von einem *LDAP*-Server verwaltet. Der Zugriff auf die Webapplikation muss jedoch auch für externe Nutzer gewährleistet werden. Da alle Nutzer des *DFN-AAI* Zugriff auf die Webapplikation erhalten sollen, wird eine föderierte Authentifizierung benötigt, die mit Hilfe von *Unity* umgesetzt wird (siehe Kapitel 3.9). Alle Nutzer - auch interne Mitarbeiter des *FZJ* - müssen sich auf diesem Wege beim Starten der Webapplikation authentisieren. Der Ablauf ist in Abbildung 4.5 dargestellt.

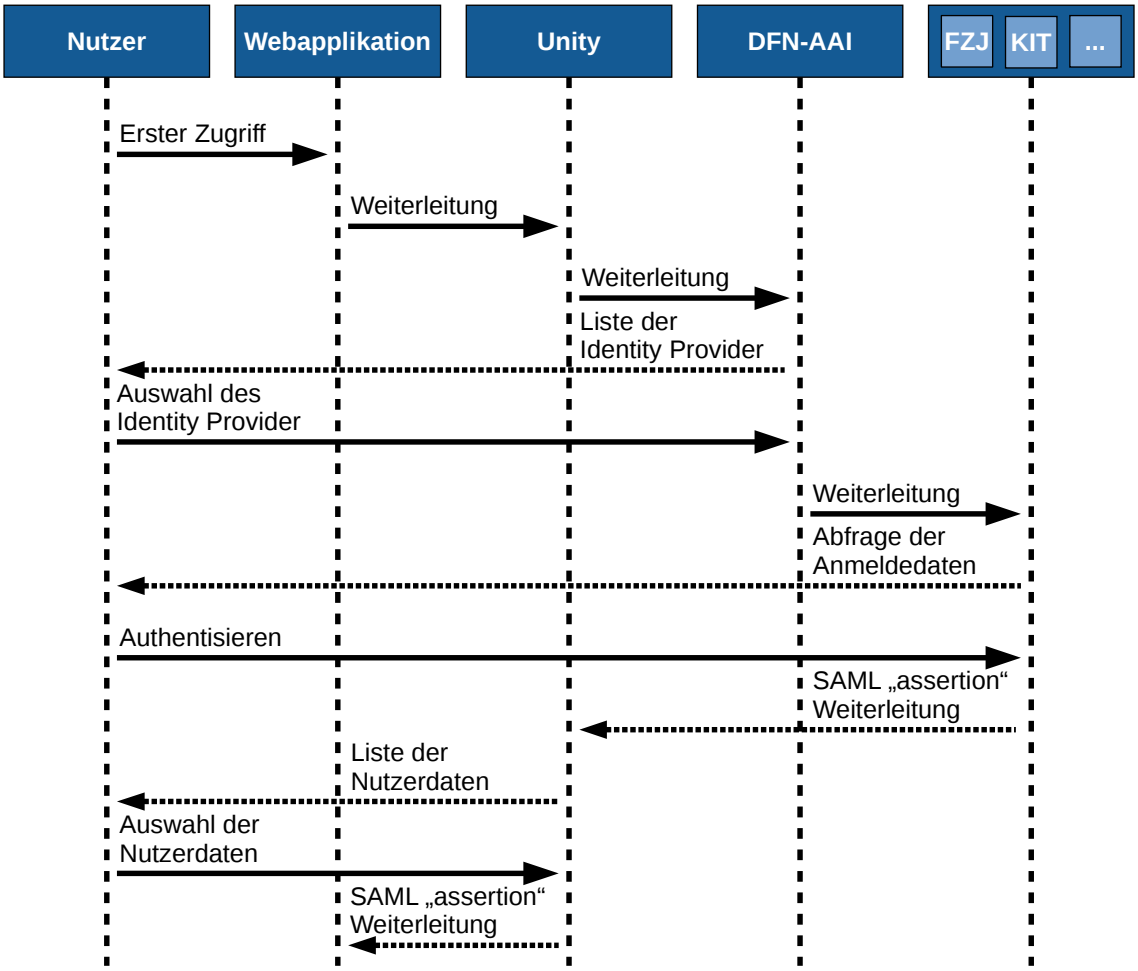

Abbildung 4.5: Ablauf der Authentisierung

Alle Nachrichten bzw. Informationen werden mittels *SAML* übermittelt.

Beim ersten Zugriff des Nutzers auf die Webapplikation wird der Nutzer auf einen *Unity*-Webserver weitergeleitet. *Unity* leitet den Nutzer dann wiederum weiter auf eine Website des *DFN-AAI*.

Abbildung 4.6: Auswahl eines Identity Provider durch die *DFN-AAI*

Hier kann der Nutzer auswählen, über welchen Identity Provider er sich anmelden möchte. Ihm werden alle Institutionen vorgeschlagen, welche Mitglieder des *DFN-AAI* sind. Zum Beispiel ein interner Nutzer aus dem *JSC* würde hier das *FZJ* als Institution auswählen. Sobald der Nutzer eine Institution ausgewählt hat, wird er auf deren Website weitergeleitet. Hier muss sich der Nutzer nun authentisieren. Dies geschieht nach den Richtlinien der ausgewählten Institution. Im *FZJ* geschieht die Authentifizierung z.B. gegenüber der lokalen Benutzerverwaltung mit Nutzername und Passwort (siehe Abbildung 4.7).

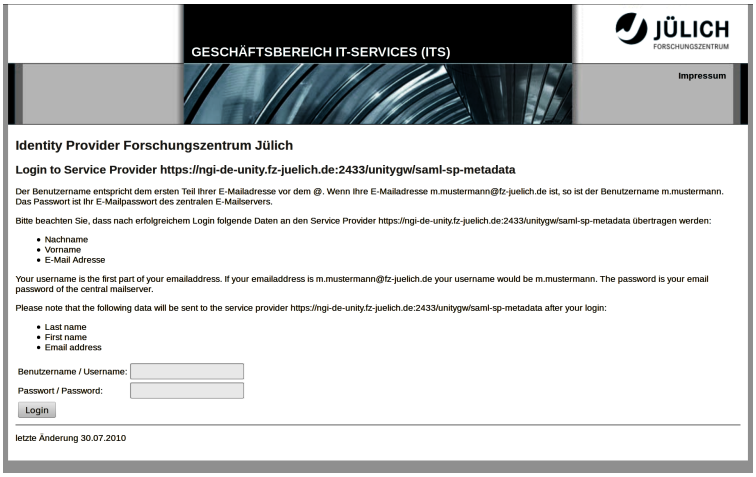

Abbildung 4.7: Identity Provider *Forschungszentrum Jülich*

Die erfolgreiche Authentifizierung bestätigt der Identity Provider dem *Unity*-Server mittels einer signierten *SAML* „assertion“ (siehe Kapitel 3.8). Der Nutzer wird gleichzeitig zurück auf die Seite von *Unity* geleitet. Hier kann der Nutzer optional auswählen welche Nutzerinformationen an die Webapplikation übermittelt werden sollen. Anschließend leitet

Unity den Nutzer auf die Webapplikation zurück und übergibt dieser die *SAML* „assertion“. Nach der erfolgreichen Authentifizierung besitzt die Webapplikation somit eine *SAML* „assertion“, welche den Nutzer eindeutig identifiziert.

Voraussetzung für diese Authentifizierung ist, dass die Webapplikation dem *Unity*-Server vertraut. Da die *SAML* „assertions“ signiert werden, kann die Webapplikation diese sicher validieren. Gleichzeitig muss ein Vertrauensverhältnis zwischen dem *Unity*-Server und der *DFN-AAI* bestehen. Durch diese Beziehung können neue Einrichtungen ins *DFN-AAI* eintreten, ohne dass dies *Unity* mitgeteilt werden muss.

4.4.4 Sichere Autorisierung

Nach der Authentisierung ist der Nutzer gegenüber der Webapplikation eindeutig identifiziert. Im nächsten Schritt muss der Nutzer nun autorisiert werden. Hierzu muss überprüft werden ob der Nutzer ein Benutzerkonto im *LDAP*-Server des *JSC* besitzt.

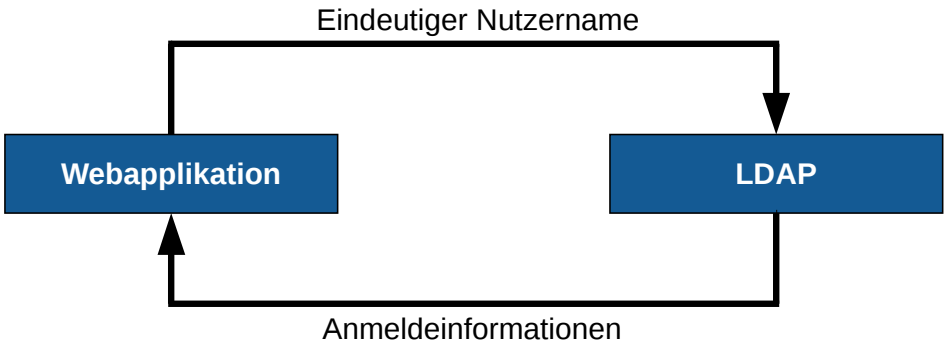

Abbildung 4.8: Autorisierung des Nutzers

Die Webapplikation gleicht die Nutzerdaten, welche sie aus der *SAML* „assertion“ liest, gegenüber dem *LDAP*-Server des *JSC* ab (siehe Abbildung 4.8). Zur Validierung wird der eindeutige Nutzername verwendet, den der Nutzer im *DFN-AAI* besitzt. Dies ist z.B. im *FZJ* die E-Mail-Adresse. So wird überprüft, ob der Nutzer ein Konto in der *LDAP*-Datenbank besitzt. Ist dies der Fall, übermittelt der *LDAP*-Server die Anmeldeinformationen (User-ID und Group-ID) an die Webanwendung. Mit diesen IDs meldet sich die Applikation anschließend beim *UFTPD* an und startet die Nutzersitzung (siehe Kapitel 4.1.3).

Die Freigaben werden nicht vom *LDAP*-Server, sondern durch die Webapplikation verwaltet. Sobald ein Nutzer eine seiner Dateien oder ein Verzeichnis freigibt, speichert die Webapplikation diese Freigabe ab. In einer Freigabe werden unter anderem der absolute Pfad des Verzeichnisses bzw. der Datei, ihre Größe und das Datum der letzten Änderung gespeichert (siehe Kapitel 4.2.1).

Weiterhin wird zu jeder Freigabe auch der Besitzer der Datei bzw. des Verzeichnisses mit

seiner User-ID und Group-ID gespeichert. Sobald nun ein Nutzer auf eine ihm freigegebene Datei zugreifen möchte, liest die Webanwendung die IDs des Besitzers aus und meldet sich mit diesen beim *UFTPD* an (siehe Kapitel 4.1.4).

Außerdem wird zu jeder Freigabe auch der eindeutige Nutzername des Nutzers, welcher die Freigabe erhält, gespeichert. Da hier der eindeutige Nutzername zur Identifikation benutzt wird, ist es auch möglich einem Nutzer Daten freizugeben, welcher keine User-ID bzw. Group-ID auf *JUST* besitzt. Bei der Anmeldung eines Nutzers gleicht die Applikation den Nutzernamen mit den gespeicherten Freigabeinformationen ab und überprüft, ob dem Nutzer Daten freigegeben wurden (siehe Kapitel 4.1.3).

Die Nutzer, welche im *LDAP* gespeichert sind, bekommen Zugriff auf ihr Heimatverzeichnis. Somit ist sichergestellt, dass Nutzer nur die für sie autorisierten Daten sehen. Ein angemeldeter Nutzer ohne *LDAP*-Konto sieht entweder einen leeren Dateibaum, oder nur die ihm freigegebenen Daten.

4.4.5 Prüfung der Nutzereingaben

Jede Eingabe des Nutzers ist eine potentielle Gefahrenstelle. Es muss verhindert werden, dass Angreifer durch gezielte Manipulation der Eingabe das System attackieren.

Jede Klartexteingabe stellt ein mögliches Angriffsziel dar. Durch die Verschlüsselung mittels *TLS,* wird die Eingabe verschlüsselt übertragen und ein Angreifer kann die Nutzerdaten nicht abfangen. Ein Angreifer könnte jedoch durch das Injizieren von Schadcode das System attackieren.

Eine mögliche Gefahrenstelle ist die Nutzeranmeldung. Hierbei muss darauf geachtet werden, dass der Angreifer z.B. keinen *SQL*-Code injizieren kann. Dadurch könnte der Angreifer frei auf die Nutzerdatenbank zugreifen und sie manipulieren. Um einem solchen Angriff entgegenzuwirken, muss die Eingabe des Nutzers zunächst validiert werden.

Auch bei der Benennung von Dateien und Verzeichnissen muss die Eingabe des Nutzers normalisiert werden. Absolute bzw. relative Pfadangaben dürfen nicht akzeptiert werden. Die meisten Angriffe dieser Art werden vom *UFTPD* abgefangen, da dieser nur mit den Rechten des angemeldeten Nutzers auf die Daten zugreift. Es könnte aber z.B. passieren, dass ein Nutzer „B“ einem Angreifer „A“ eines seiner Verzeichnisse freigibt, z.B. das Verzeichnis „`~/data/`“. Der Angreifer „A“ könnte nun eine Datei mit dem Namen „`~/.ssh/id_rsa`“ in das Verzeichnis des Nutzers „B“ ablegen. Durch diesen absoluten Pfad würde der Angreifer also auf ein ihm nicht freigegebenes Verzeichnis zugreifen. Die Webapplikation meldet sich in diesem Fall als Nutzer „B“ beim *UFTPD* an. Der Zugriff geschieht für den *UFTPD* also unter dem Nutzerkonto des Besitzers der Datei und somit würde er den Zugriff akzeptieren. So könnte der Angreifer auf den privaten *SSH*-Schlüssel („`id_rsa`“) des Nutzers „B“ zugreifen. Die Normalisierung der Nutzereingabe verhindert solche Angriffe, da absolute Pfadangaben nicht akzeptiert werden.

4.4.6 Sicherheit vor Datenverlust

Alle Zugriffe des Nutzers auf seine Daten werden mittels *UFTP* direkt an *JUST* weitergeleitet. Um den Nutzer vor Datenverlust zu schützen bzw. damit er nicht ungewollt sein System beschädigt, muss die Webapplikation vor kritischen Aktionen nachfragen, ob der Nutzer die Aktion wirklich durchführen möchte.

Folgende Dateizugriffe werden als kritisch eingestuft und der Nutzer muss die Ausführung explizit bestätigen:

- Löschen eines Verzeichnisses
- Löschen einer Datei
- Überschreiben einer Datei
- Freigabe eines Verzeichnisses

Zusätzlich bietet *JUST* durch redundante Datenspeicherung und regelmäßige Backups Sicherheit vor Datenverlust.

Alle Daten werden in der Reed-Solomon Kodierung[2] gespeichert. Dieses Fehlerkorrekturverfahren bietet bei möglichst wenig Redundanz eine hohe Ausfall- und Fehlertoleranz. Die Daten werden nach dem Muster $8 + 2p$ abgespeichert, das heißt auf acht Datenblöcke kommen zwei Paritätsblöcke, dadurch können bis zu zwei Fehler korrigiert bzw. ausgeglichen werden.

Die Metadaten werden zusätzlich noch einem *RAID 10* Verbund gespeichert. Das heißt die Reed-Solomon kodierten Daten werden gespiegelt und verteilt.

Als Backup der Daten wird von jeder Datei im Dateisystem eine Kopie auf den Bandlaufwerken abgelegt. Sobald die Datei geändert wird, wird die neue Version gesichert. Die alte Version wird als „inaktiv“ gekennzeichnet, und kann vom Nutzer wiederhergestellt werden. Es werden bis zu vier „inaktive“ Versionen vorgehalten. Die letzte inaktive Version wird immer 120 Tage, alle anderen 30 Tage vorgehalten. Somit können auch versehentlich gelöschte Daten auf einen Wiederherstellungspunkt in der Vergangenheit zurückgesetzt werden.

Des Weiteren werden die Daten, wie bereits beschrieben, zusätzlich durch eine Ende-zu-Ende Prüfsumme gesichert. Hierbei können sowohl Festplattenfehler als auch Fehler, welche während der Datenübertragung entstanden sind, erkannt und behoben werden.

[2] Die Reed-Solomon Kodierung ist ein um 1960 von Irving S. Reed und Gustave Solomon entwickeltes Fehlerkorrekturverfahren.

5 Realisierung

In diesem Kapitel wird beschrieben, wie der im vorherigen Kapitel konzipierte Softwareentwurf realisiert wurde. Hierfür werden Code Beispiele aufgelistet und beschrieben. Außerdem wird die entwickelte grafische Weboberfläche in Abbildungen vorgestellt.

Zur Verwaltung und Standardisierung des Projektes wird *Maven* verwendet. Dadurch wird die Entwicklungszeit sehr verkürzt. *Maven* packt das Projekt, löst alle Abhängigkeiten auf, führt Tests durch und startet den Server. Auf diese Weise kann die Webanwendung schnell und ohne viel Aufwand online getestet werden. Ein weiterer Vorteil von *Maven* ist, dass es das gesamte Projekt während der Entwicklung regelmäßig scannt. Sobald es Änderungen findet, wird das Projekt, während der Server noch läuft, neu installiert. So werden dem Entwickler Änderungen im Code sofort sichtbar gemacht (siehe Kapitel 3.6).

5.1 Kommunikation über *UFTP*

Der Zugriff auf die Daten der Dateisysteme geschieht mittels *UFTP*. Wie in Kapitel 4.1.3 beschrieben, wird zunächst eine *UFTP*-Sitzung unter dem angemeldeten Nutzer gestartet, falls der Nutzer ein Benutzerkonto auf dem Dateisystem hat. Für die freigegebenen Dateien und Verzeichnisse wird jeweils gruppiert, eine *UFTP*-Sitzung unter dem Nutzernamen des Besitzers gestartet. Zur Schonung der Systemressourcen werden diese Sitzungen allerdings nur temporär aufgebaut und nach dem Dateizugriff wieder geschlossen.

5.1.1 Starten einer *UFTP*-Sitzung

Zum Start der *UFTP*-Sitzung muss zunächst ein Objekt der Klasse `UFTPTransferRequest` erstellt werden. Dieses beschreibt einen Datentransfer zwischen dem Client und dem Server. Der Konstruktor der Klasse erstellt das Objekt und erwartet hierzu folgende Argumente:

```
1 UFTPTransferRequest uftpTransferRequest = new
                  ↪UFTPTransferRequest(client, send, numCons,
                  ↪ file, append, secret, user, group, key);
```

Abbildung 5.1: Erstellung eines `UFTPTransferRequest`

- `client : java.net.InetAddress`
  Die *IP*-Adresse des *UFTP*-Clients, also in diesem Fall die *IP*-Adresse des Webservers, da dieser mit dem *UFTP*-Server kommuniziert.

- `send : boolean`
  Dieser Parameter gibt an, ob die Datei gesendet oder empfangen wird. Da eine neue Sitzung gestartet wird, in der sowohl Daten gesendet, als auch empfangen werden können, ist die Wahl dieses booleschen Parameters unerheblich.

- `numCons : int`
  Dieser Parameter gibt an, wie viele Datenverbindungen parallel geöffnet werden können.

- `file : java.io.File`
  Standardmäßig ist das `File` der Pfad der Datei, auf die zugegriffen werden soll. Um eine Sitzung zu starten, wird ein Objekt der Klasse `File` mit einer vorher definierten Zeichenkette angelegt:

```
1 File file = new File(UFTPWorker.sessionModeTag);
```

Abbildung 5.2: Anlegen einer speziellen Datei zum Starten einer *UFTP*-Sitzung

  Der String `UFTPWorker.sessionModeTag` ist eine feste Zeichenkette, die dem *UFTP*-Server anzeigt, dass eine neue Sitzung gestartet wird. Das aktuelle Verzeichnis einer Sitzung ist beim Start das Heimatverzeichnis des Nutzers.

- `append : boolean`
  Dieser Parameter gibt an ob an die Datei angehängt oder überschrieben wird. Da jedoch eine *UFTP*-Sitzung gestartet wird, ist auch dieser Parameter unbedeutend.

- `secret : String`
  Das „Geheimnis“ dient der Identifikation gegenüber dem *UFTP*-Server (siehe Kapitel 3.3.6).

- `user : String`
  Der Name des Nutzers.

- `group : String`
  Der Name der Nutzergruppe des Nutzers.

- `key : byte[]`
  Ein symmetrischer Schlüssel für die Verschlüsselung der Daten (siehe Kapitel 3.3.6).

Das Objekt der Klasse `UFTPTransferRequest` wird nach dem Erstellen an den Server gesendet:

```
1 uftpTransferRequest.sendTo(server, jobPort);
```

Abbildung 5.3: Senden eines `UFTPTransferRequest` an den Server

Die Methode erwartet zum Senden folgende Parameter:

- `server : java.net.InetAddress`
  Die *IP*-Adresse des *UFTP*-Servers.
- `jobPort : int`
  Der „command“ Port über den der *UFTP*-Server gesteuert wird (siehe Kapitel 3.3.1).

Ist die Dateianfrage an der Server gesendet worden, startet die *UFTP*-Sitzung serverseitig. Das heißt, der *UFTP*-Server akzeptiert nun alle Dateizugriffe des Nutzers auf Dateien, die ihm gehören.

Nachdem die *UFTP*-Sitzung beim Server gestartet wurde, muss ein `UFTPSessionClient` erstellt werden.

```
1 UFTPSessionClient uftpSessionClient = new UFTPSessionClient(
      ↪server, dataport);
```

Abbildung 5.4: Anlegen eines `UFTPSessionClient`

Dieser Client stellt Methoden für den Datenzugriff zur Verfügung. Die Methoden werden im Kapitel 5.1.2 aufgelistet. Der Konstruktor des `UFTPSessionClient` erwartet folgende Argumente:

- `server : java.net.InetAddress`
  Die *IP*-Adresse des *UFTP*-Servers.
- `dataPort : int`
  Der „listen“ Port des *UFTP*-Servers, über diesen werden die Dateien versendet.

Nach dem Erstellen des `UFTPSessionClient` muss das „Geheimnis“, welches beim Starten der Sitzung gewählt wurde, gesetzt werden.

```
1 uftpSessionClient.setSecret(secret);
```

Abbildung 5.5: Setzen des „Geheimnis“ beim `UFTPSessionClient`

Anschließend verbindet sich der Client mit dem Server.

```
1 uftpSessionClient.connect();
```

Abbildung 5.6: Verbindung des `UFTPSessionClient` mit dem Server

Nun ist eine *UFTP*-Sitzung gestartet worden und der Client ist mit dem Server verbunden.

5.1.2 Methoden für den Dateizugriff

Der Dateizugriff des Webservers zum *UFTP*-Server geschieht über das Objekt der Klasse `UFTPSessionClient`, dessen Erstellung im vorigen Kapitel beschrieben wurde. Das Objekt stellt unter anderem folgende Methoden zur Verfügung:

- `disconnect()`
  Beendet die *UFTP*-Sitzung und trennt die Verbindung zum Server.
- `get(remoteFile : String, localTarget : java.io.OutputStream)`
  Lädt eine Datei, deren Name in `remoteFile` steht, und schreibt sie in den übergebenen `OutputStream` namens `localTarget`.
- `put(remoteFile : String, size : long, localSource : java.io.InputStream)`
  Mit Hilfe dieser Methode werden Dateien auf den *UFTP*-Server hochgeladen. Die Zeichenkette `remoteFile` enthält den zukünftigen Namen der Datei. Diese wird neu angelegt, falls sie noch nicht existiert, ansonsten überschrieben. Die Methode liest den `InputStream` aus und schreibt den Inhalt in die Datei. Das Argument `size` gibt hierbei die Anzahl der zu schreibenden Bytes an.
- `getFileInfoList(baseDir : String) : List<FileInfo>`
  Gibt eine Liste aller Dateien bzw. Verzeichnisse zurück, welche sich in dem Verzeichnis namens `baseDir` befinden. Die Klasse `FileInfo` gehört zur Bibliothek von *UFTP*. Sie speichert den Namen, die Größe und das letzte Änderungsdatum einer Datei bzw. eines Verzeichnisses.
- `cd(dir : String)`
  Wechselt das aktuelle Verzeichnis der Sitzung.
- `mkdir(dir : String)`
  Erstellt ein neues Verzeichnis mit dem übergebenen Namen.
- `rm(path : String)`
  Löscht die übergebene Datei bzw. das Verzeichnis. Im Falle eines Verzeichnisses werden alle Unterverzeichnisse oder Dateien innerhalb dieses Verzeichnisses rekursiv gelöscht.
- `rename(from : String, to : String)`
  Benennt eine Datei um bzw. verschiebt sie von `from` nach `to`.

- `stat(path : String) : FileInfo`
  Erfragt die neuesten Informationen zu der übergebenen Datei oder dem Verzeichnis.

5.2 Implementierung der grafischen Oberfläche in *Vaadin*

Die grafische Oberfläche wurde in *Vaadin* entwickelt. Die einzigen Voraussetzungen für den Nutzer sind eine Internetanbindung und ein *HTML5* unterstützender Browser. Das bietet die Möglichkeit, die Webapplikation auf fast jedem Endgerät aufzurufen, wie z.B. auf einem Smartphone oder einem Tablet. Die Applikation passt sich jeweils der entsprechenden Bildschirmauflösung an.

5.2.1 Anmeldebildschirm der Webanwendung

Nachdem der Nutzer die URL der Webapplikation in seinen Browser eingegeben hat, gelangt er zum Anmeldebildschirm. Da die Schnittstelle zu *Unity* nicht zum Rahmen dieser Masterarbeit gehört, wurde hier ein vorläufiger Anmeldebildschirm erstellt. Dieser besteht aus zwei Eingabefeldern: Eins zur Eingabe des Nutzernamens und eins zur Eingabe des Passworts (siehe Abbildung 5.7). Die durch rote Sternchen gekennzeichneten Pflichtfelder müssen ausgefüllt werden.

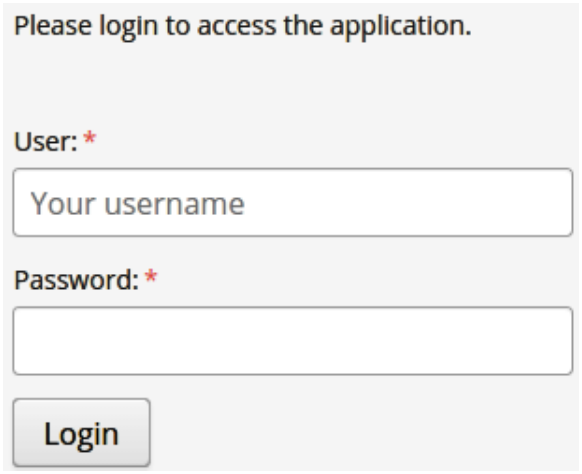

Abbildung 5.7: Anmeldebildschirm

5.2.2 Oberfläche in Form der `TreeTable`

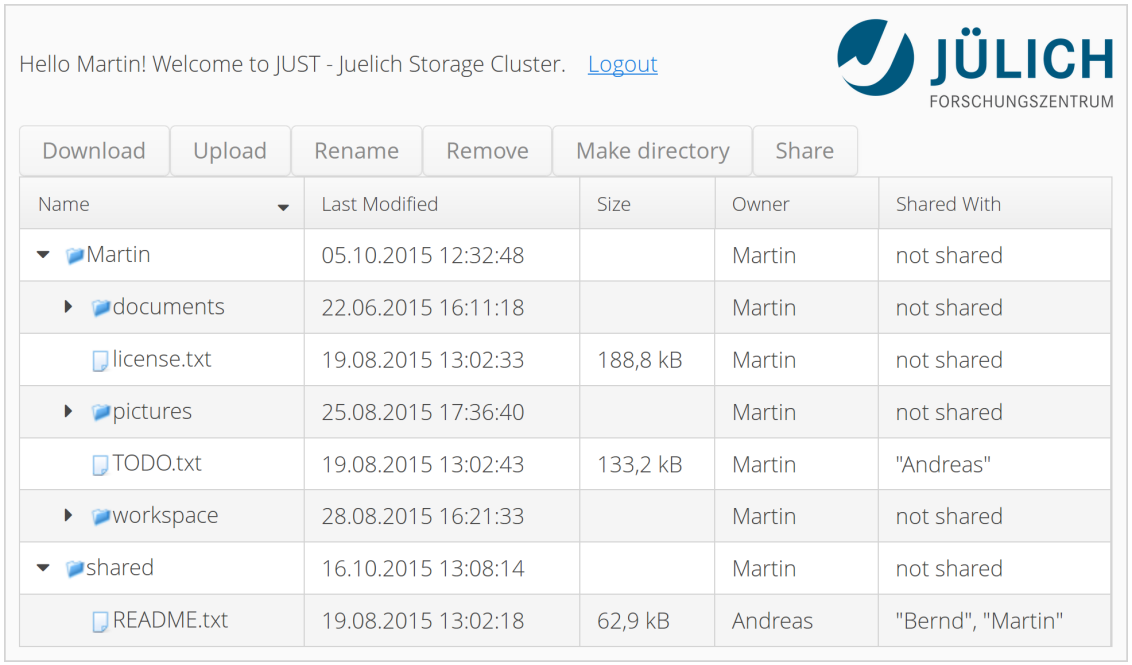

Abbildung 5.8: Hauptbildschirm der Applikation

Nach der Anmeldung gelangt der Nutzer auf den Hauptbildschirm der Applikation (siehe Abbildung 5.8). Hier werden ihm jederzeit alle Dateien angezeigt. Das Kernstück der Oberfläche ist die `TreeTable`. Die `TreeTable` ist eine grafische Komponente aus *Vaadin*. Die Komponente vereint die Funktionen einer Tabelle und eines Verzeichnisbaums. Abgebildet ist die für das Projekt angepasste `TreeTable` in Abbildung 5.9.

Name	Last Modified	Size	Owner	Shared With
Martin	05.10.2015 12:32:48		Martin	not shared
documents	22.06.2015 16:11:18		Martin	not shared
license.txt	19.08.2015 13:02:33	188,8 kB	Martin	not shared
pictures	25.08.2015 17:36:40		Martin	not shared
TODO.txt	19.08.2015 13:02:43	133,2 kB	Martin	"Andreas"
workspace	28.08.2015 16:21:33		Martin	not shared
shared	09.10.2015 09:50:01		Martin	not shared
README.txt	19.08.2015 13:02:18	62,9 kB	Andreas	"Bernd", "Martin"

Abbildung 5.9: Beispiel für einen Dateibaum

Die Dateien werden in einem Verzeichnisbaum dargestellt. Ein Verzeichnisbaum ist intuitiv und den meisten Nutzern bereits bekannt. Somit ist die Anforderung der Benutzerfreundlichkeit erfüllt.

Verzeichnisse können beliebig aufgeklappt und auch wieder geschlossen werden. Ist ein Verzeichnis aufgeklappt, werden seine Dateien und Unterverzeichnisse, eingerückt unter dem Namen des Verzeichnisses, aufgelistet. Nur für die aktuell angezeigten Dateien bzw. Verzeichnisse werden Informationen vom *UFTP*-Server erfragt. Das heißt, die Dateien und Unterverzeichnisse nicht aufgeklappter Verzeichnisse werden nicht abgerufen. Jedes Element des Verzeichnisbaums hat ein kleines Icon, das den Dateityp kennzeichnet. Hierzu werden die Standard-Icons verwendet, welche aus diversen Betriebssystemen bereits bekannt sind. Das erhöht die Benutzerfreundlichkeit weiter. Zum Beispiel wird ein Verzeichnis durch eine Aktenmappe symbolisiert.

Der Verzeichnisbaum ist integriert in eine Tabelle, die neben dem Dateinamen bzw. dem Verzeichnisnamen in der ersten Spalte, in weiteren Spalten weitere Informationen anzeigt. Dazu gehören das letzte Änderungsdatum, die Größe und der Besitzer. Weiterhin wird auch angezeigt ob die Datei freigegeben wurde. Wenn sie entsprechend freigegeben wurde, stehen in dieser Spalte, anstatt „not shared“, die Namen, bzw. die Gruppe der Benutzer mit denen die Datei geteilt wurde.

Die Daten werden in zwei Wurzelverzeichnisse eingeteilt. Das erste Wurzelverzeichnis stellt das Heimatverzeichnis dar. Dieses sehen nur die Nutzer, die auch ein Nutzerprofil auf dem Dateisystem besitzen. In dem Heimatverzeichnis werden alle Dateien des angemeldeten Nutzers aufgelistet. Im zweiten Wurzelverzeichnis, namens „shared“, werden dem angemeldeten Nutzer alle Dateien angezeigt, welche ihm von anderen Nutzern freigegeben wurden.

Die dargestellte `TreeTable` kann nach verschiedenen Kriterien aufsteigend und absteigend sortiert werden. Standardmäßig wird die Tabelle aufsteigend nach Namen sortiert. Durch einen Klick auf eine Spaltenüberschrift wird die Tabelle nach diesem Kriterium sortiert. Klickt der Nutzer z.B. auf die Spaltenüberschrift „Last Modified“, so werden die Dateien nach ihrem letzten Änderungsdatum sortiert. Die hierarchische Struktur wird hierbei immer beachtet, so dass die Dateien nur innerhalb eines Verzeichnisses sortiert werden.

Mittels „Drag and Drop“ können Dateien und Verzeichnisse verschoben werden. Die Operation wird mit der in Kapitel 5.1.2 beschriebenen Methode `rename` an den *UFTP*-Server weitergeleitet und dort sofort ausgeführt.

Die Informationen über die Daten des Nutzers bekommt die `TreeTable` von dem Container `IContainer` (siehe Kapitel 4.3.2). Der Container wird wie folgt an die `TreeTable` gebunden:

```
1 treeTable.setContainerDataSource(iContainer);
```

Abbildung 5.10: Bindung des Containers an die `TreeTable`

Durch diese Bindung erhält die `TreeTable` die benötigten Informationen zum Anzeigen.

Außerdem benachrichtigt der Container die `TreeTable` bei Änderungen an den Daten, so dass sich die grafische Komponente entsprechend aktualisieren kann.

5.2.3 Funktionalitäten der Oberfläche

Es gibt sechs Schaltflächen zur Steuerung der `TreeTable` und zur Datenverarbeitung. Diese sind in Abbildung 5.11 dargestellt.

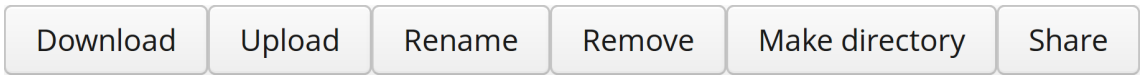

Abbildung 5.11: Schaltflächen der Applikation

Nach der Anmeldung des Nutzers sind alle Schaltflächen zunächst deaktiviert. Erst nachdem der Nutzer per Mausklick eine Zeile aus der `TreeTable` auswählt, werden die entsprechenden Schaltflächen aktiviert. Deren Funktionen werden im Folgenden genauer erläutert.

5.2.3.1 Funktionalität „Download“

Die Schaltfläche „Download“ initiiert das Herunterladen der aktuell in der `TreeTable` ausgewählten Datei. Solange der Nutzer noch keine Datei ausgewählt hat, ist diese Schaltfläche deaktiviert.

Sobald die Schaltfläche angeklickt wurde, fragt die Webapplikation die ausgewählte Datei beim *UFTP*-Server an und initiiert den Datenaustausch.

Vaadin leitet die Datei vom Webserver an den Browser des Nutzers weiter. Hierzu wird die von *Vaadin* bereitgestellte Komponente `FileDownloader` verwendet. Der `FileDownloader` übernimmt die Client-Server Kommunikation. Er wird an einen `Button` gebunden und startet das Herunterladen, sobald der `Button` angeklickt wird. Der `FileDownloader` benötigt einen `InputStream`, aus dem er die Daten lesen kann. Hierfür definiert *Vaadin* das Interface `StreamSource`:

```
1 public interface StreamSource extends Serializable {
2    public InputStream getStream();
3 }
```

Abbildung 5.12: Das Interface `StreamSource`

Das Interface `StreamSource` gehört zu der Bibliothek von *Vaadin*. Es ist die Schnittstelle zu den Daten, und übergibt den `InputStream` an den `FileDownloader`.

Zum Abrufen der Datei beim *UFTP*-Server wird die vom `UFTPSessionClient` bereitgestellte Methode `get` verwendet (siehe Kapitel 5.1.2). Diese Methode erwartet einen `OutputStream` als Übergabeparameter, in den die Daten geschrieben werden. Wie bereits erwähnt, benötigt die „Download“ Komponente von *Vaadin* jedoch einen `InputStream`, von dem sie die Daten lesen kann. Das bedeutet, der `OutputStream` der Methode `get` muss in einen `InputStream` geleitet werden. Hierzu werden die in *Java* vorhandenen Klassen `PipedInputStream` und `PipedOutputStream` verwendet. Der folgende Programmausschnitt zeigt, wie dies realisiert wurde:

```
1  final PipedInputStream input = new PipedInputStream();
2  final PipedOutputStream output = new PipedOutputStream();
3  output.connect(input);
4  new Thread(() -> {
5      try {
6          uftpSessionClient.get(path, output);
7      } catch (IOException e) {
8          LOGGER.error("IOException␣at␣downloading␣file:␣" +
                   ↪path, e);
9      } finally {
10         output.close();
11     }
12 }).start();
```

Abbildung 5.13: Direkte Umleitung eines `OutputStream` in einen `InputStream`

In den ersten beiden Zeilen werden zunächst Instanzen der Klassen `PipedInputStream`, bzw. `PipedOutputStream` initiiert. In der dritten Zeile werden die beiden Streams miteinander verbunden. An dieser Stelle kann eine `IOException` geworfen werden, diese wird abgefangen und entsprechend aufgezeichnet. In Zeile vier wird anschließend ein neuer Thread erstellt. Dieser wird benötigt, da während dem Empfangen der Datei die empfangenen Daten direkt weitergeleitet werden. So müssen die Daten nicht auf dem Webserver zwischengespeichert werden, sondern werden direkt von dem *UFTP*-Server an den Nutzer weitergeleitet. In diesem Thread wird die Methode `get` aufgerufen (Zeile sechs), mit der die Datei vom *UFTP*-Server geholt wird. Der Aufruf der Methode `get` kann wiederum eine `IOException` hervorrufen. Diese wird auch abgefangen und aufgezeichnet. Nachdem die Datei vollständig gelesen wurde, oder ein Fehler entstanden ist, muss der `OutputStream` in Zeile zehn wieder geschlossen werden. Zum Schluss wird der Thread gestartet (Zeile 12) und der `InputStream` kann ausgelesen werden.

Durch dieses Verfahren werden die Daten ohne Umwege direkt auf dem Client gespeichert.

5.2.3.2 Funktionalität „Upload“

Die Schaltfläche „Upload“ initiiert das Hochladen einer Datei. Diese Schaltfläche wird aktiviert, wenn der Nutzer vorher in der `TreeTable` ein Verzeichnis auswählt. In das ausgewählte Verzeichnis wird die Datei gespeichert. Die Schaltfläche aktiviert sich nur, wenn der Nutzer die Rechte hat, eine Datei in das ausgewählte Verzeichnis zu schreiben. Falls das Verzeichnis ihm gehört, hat der Nutzer immer die Rechte eine Datei hochzuladen. Wenn das ausgewählte Verzeichnis jedoch dem Nutzer von einem anderen Nutzer freigegeben wurde, prüft der `ACLManager` zunächst, ob der Nutzer Schreibrechte auf das Verzeichnis hat.

Sobald der Nutzer die Schaltfläche anklickt, öffnet sich, abhängig vom Betriebssystem und Browser, ein entsprechender Datei-Browser. Dort wählt der Nutzer aus seinen lokalen Dateien die Datei zum Hochladen aus. Es kann jede Datei unabhängig vom Dateiformat hochgeladen werden. Falls in dem ausgewählten Verzeichnis bereits eine Datei mit dem selben Namen existiert, muss der Nutzer das Überschreiben der Datei explizit bestätigen. Die ausgewählte Datei wird anschließend automatisch hochgeladen. Hierzu wird die von *Vaadin* bereitgestellte Komponente `Upload` verwendet. Die Komponente benötigt einen `OutputStream`, in den sie die Datei schreiben kann.

Hierzu definiert *Vaadin* wieder eine Schnittstelle zwischen der grafischen Komponente und dem Datenmodell. Das entsprechende Interface heißt `Receiver`:

```
1 public interface Receiver extends Serializable {
2    public OutputStream receiveUpload(String filename, String
                 ↪ mimeType);
3 }
```

Abbildung 5.14: Das Interface `Receiver`

Der `OutputStream` wird der `Upload` Komponente durch den `Receiver` zur Verfügung gestellt. Die Methode `receiveUpload` bekommt den Namen der Datei und den *MIME-Type*[1] von der `Upload` Komponente übergeben und gibt den `OutputStream` zurück. Anhand des *MIME-Type* wird das entsprechende Icon zur Darstellung ausgewählt.

Ähnlich wie beim „Download“ entsteht zunächst durch die Methode `put` dieselbe Problematik wie bei der Methode `get`. Denn die Methode `put` (Zeile 6), mit der Dateien zum *UFTP*-Server hochgeladen werden, benötigt diesmal einen `InputStream`, von dem die Datei gelesen werden kann. Wie bereits beschrieben, erwartet die `Upload` Komponente von *Vaadin* jedoch einen `OutputStream`, in den sie die Datei schreiben kann. Der `OutputStream` muss wieder in den `InputStream` umgeleitet werden (siehe Abbildung 5.15):

[1] Der MIME-Type (**M**ultipurpose **I**nternet **M**ail **E**xtensions) gibt an in welcher Form die Daten gespeichert sind. Daten können z.B. in Klartext, als Bild, als ein *HTML*-Dokument oder in vielen anderen Typen gesendet werden.

```
final PipedOutputStream output = new PipedOutputStream();
final PipedInputStream input = new PipedInputStream();
output.connect(input);
new Thread(() -> {
    try {
        uftpSessionClient.put(path, size, input);
    } catch (IOException e) {
        LOGGER.error("IOException at uploading file: "+path, e
            );
    } finally {
        input.close();
    }
}).start();
```

Abbildung 5.15: Direkte Umleitung eines `InputStream` in einen `OutputStream`

Der entsprechende Code ähnelt sehr dem in Abschnitt 5.2.3.1 abgebildeten Code zum Herunterladen einer Datei. Einziger Unterschied ist, dass im neu erstellten Thread nun der `InputStream` ausgelesen wird.

5.2.3.3 Funktionalität „Rename“

Mit Hilfe der Schaltfläche „Rename“ können Dateien und Verzeichnisse umbenannt werden. Die Schaltfläche wird aktiviert sobald der Nutzer eine Datei oder ein Verzeichnis mit entsprechenden Schreibrechten auswählt. Der Nutzer benötigt zum Umbenennen Schreibrechte auf das der Datei übergeordnete Verzeichnis. Dateien und Verzeichnisse, die der Nutzer freigegeben bekommen und auf die er nur Leserechte hat, darf er somit nicht umbenennen. Mit einem Klick auf diese Schaltfläche wird der Name des aktuell ausgewählten Elements editierbar:

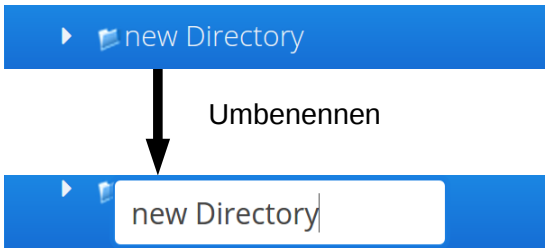

Abbildung 5.16: Umbenennen einer Datei

Der Nutzer kann den Namen nun beliebig ändern. Durch Drücken der Enter-Taste wird das Umbenennen bestätigt. Die Aktion wird direkt an den *UFTP*-Server weitergeleitet. Hierzu wird die in Kapitel 5.1.2 vorgestellte Methode `rename` verwendet.

5.2.3.4 Funktionalität „Remove“

Mit einem Klick auf die Schaltfläche „Remove“ löscht der Nutzer die in der `TreeTable` ausgewählte Datei bzw. das ausgewählte Verzeichnis. Die Schaltfläche aktiviert sich nur, wenn der Nutzer die Rechte hat, die Datei bzw. das Verzeichnis zu löschen. Dies ist dann der Fall, wenn er Schreibrechte auf das, der Datei übergeordnete, Verzeichnis besitzt. Vor dem Löschen bekommt der Nutzer eine Warnmeldung angezeigt und muss den Vorgang explizit bestätigen. Die Operation wird anschließend mittels der in Kapitel 5.1.2 beschriebenen Methode `rm` an den *UFTP*-Server weitergeleitet und dort sofort ausgeführt.

5.2.3.5 Funktionalität „Make directory“

Durch das Betätigen der Schaltfläche „Make directory“ wird ein neues Verzeichnis in dem aktuell ausgewählten Verzeichnis erstellt. Die Schaltfläche ist nur aktiv, falls der Nutzer in der `TreeTable` ein Verzeichnis ausgewählt hat, auf das er Schreibrechte besitzt. Zum Erstellen des Verzeichnisses wird die Methode `mk` des `UFTPSessionClient` verwendet. Das neue Verzeichnis wird automatisch „new directory“ genannt. Falls schon ein Verzeichnis mit diesem Namen im gleichen Verzeichnis existiert, wird das neue Verzeichnis durchnummeriert.

5.2.3.6 Funktionalität „Share“

Über die Schaltfläche „Share“ werden die Freigaben verwaltet. Zunächst muss der Nutzer eine Datei oder ein Verzeichnis aus der `TreeTable` auswählen. Mit einem Klick auf die Schaltfläche öffnet sich ein neues Fenster (siehe Abbildung 5.17).

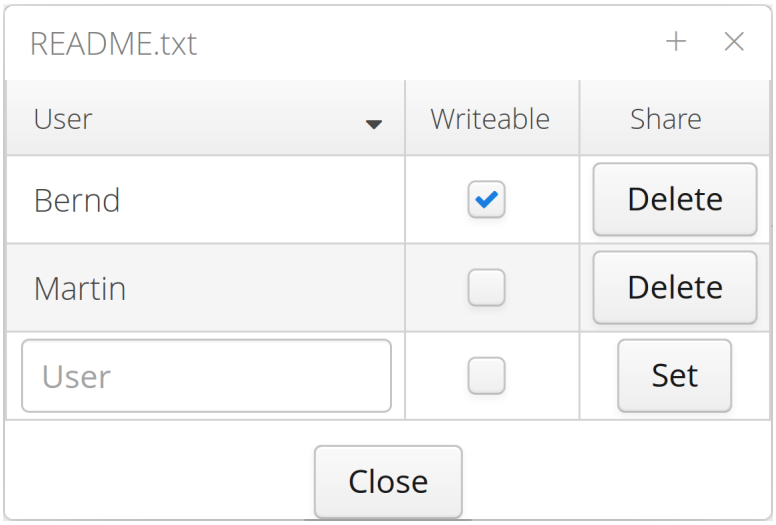

Abbildung 5.17: Freigaben-Fenster

In der Titelleiste steht der Name der ausgewählten Datei bzw. des ausgewählten Verzeichnisses. In der Abbildung 5.17 wurde die Datei „README.txt“ ausgewählt.
Unter der Titelleiste werden nun alle Freigaben der Datei tabellarisch aufgelistet. Die Datei „README.txt“ ist für den Nutzer „Bernd“ und den Nutzer „Martin“ freigegeben. Die Tabelle kann nach den Namen der „User“ aufsteigend und absteigend sortiert werden. „Martin“ darf die Datei nur lesen, jedoch nicht schreiben. Dies wird durch die Kontrollkästchen in der zweiten Spalte gekennzeichnet. Ein Häkchen in dem Kästchen zeigt an, dass der entsprechende „User“ auch Schreibrechte besitzt. Falls der Nutzer der Besitzer der ausgewählten Datei ist, kann er durch ein Klicken auf das Kontrollkästchen Schreibrechte vergeben und auch wieder aufheben. Die Schaltfläche „Delete“ in der dritten Spalte löscht den Eintrag aus der Liste. Dadurch sind immer nur die Nutzer aufgelistet, für welche die Datei aktuell freigegeben ist. Nur der Besitzer einer Datei bzw. eines Verzeichnisses kann Freigaben für andere Nutzer löschen. Ein Nutzer, dem eine Datei oder ein Verzeichnis freigegeben wurde, kann nur die Freigabe löschen, die ihn selber betrifft. Die restlichen Schaltflächen sind in diesem Fall deaktiviert.
In der letzten Zeile der Tabelle kann der Besitzer einer Datei neue Freigaben für andere Nutzer setzen. Hierzu tippt er den Namen des Nutzers in das Textfeld. Anschließend wählt er über das Kontrollkästchen mit der Beschriftung „Writeable“ aus, ob auch Schreibrechte vergeben werden sollen. Mit einem Klick auf die Schaltfläche „Set“ wird die Freigabe gesetzt, und eine neue Zeile wird zur Tabelle hinzugefügt.

6 Evaluation

6.1 Test der Anwendung

Während der Entwicklung wurde ein *UFTP*-Testserver verwendet. Die Webapplikation wurde mehrfach auf diesem Testserver mit Testdaten getestet. Durch diese simulierte Umgebung konnte die Applikation autark auf einem lokalen Rechner getestet werden und Fehler konnten somit das laufende System nicht beeinflussen.

Zum Testen des Zugriffs auf „fremde“, geteilte Dateien und Verzeichnisse wurden mehrere Nutzer angelegt. Von diesen wurden verschiedene Freigaben erstellt und der Zugriff getestet. Die Anzeige der Webapplikation stimmt mit dem tatsächlichen Dateibaum überein. Berechtigungen werden korrekt übernommen.

Die Black-Box- und White-Box-Tests verliefen fehlerfrei, so dass die Webapplikation auf *JUDAC* für einen Testbetrieb installiert wird. Somit können ausgewählte Endnutzer die Webapplikation nutzen.

6.2 Soll-Ist-Vergleich

Die in Kapitel 2.2 an das Projekt gestellten Anforderungen wurden erfolgreich umgesetzt.

SK1: “Jeder Nutzer von *JUST* soll Zugriff auf seine Daten erhalten.“

Die Applikation ist mittels *UFTP* mit *JUST* verbunden, dadurch ist gewährleistet, dass jeder Nutzer auf seine Daten zugreifen kann.

SK2: “Ein Nutzer soll anderen Nutzern Zugriff auf seine Dateien gewähren können.“

Durch die in Kapitel 5.2.3.6 beschriebene Funktionalität „share“, können Nutzer ihre Dateien und Verzeichnisse anderen Nutzern, mit verschiedenen Berechtigungen, zur Verfügung stellen. Zusätzlich kann jederzeit überprüft werden, wem die Daten freigegeben wurden. Die Freigaben können auch leicht zurückgenommen werden.

SK3: “Der Besitz der Daten darf nicht an Dritte abgegeben werden.“

Da die Daten auf *JUST* liegen bleiben, wird der Besitz der Daten nicht abgegeben.

SK4: “Die Struktur der Dateisysteme soll nicht geändert werden.“

Da *UFTP* mit verschiedensten Dateisystemen interagieren kann, ist die Webapplikation unabhängig vom Speicher. Die Struktur der Dateisysteme muss also nicht geändert werden.

SK5: "Die Applikation soll sicher sein und ein vertrauenswürdiges Umfeld bieten."

Die Sicherheit der Applikation wird durch die Verwendung sicherer Protokolle wie *TLS* und *UFTP* gewährleistet. Durch den Zugriff über *Unity* befindet sich der Nutzer in einer vertrauten Umgebung.

SK6: "Die Daten sollen nicht auf ein weiteres Dateisystem transferiert werden müssen."

Die Daten werden mittels *UFTP* nur auf Anfrage von *JUST* abgerufen und werden nicht auf ein anderes Dateisystem transferiert. Alle Datenoperationen werden direkt auf *JUST* ausgeführt und müssen nicht zwischengespeichert werden.

SK7: "Sicherheitsrichtlinien sollen nicht geändert werden."

Die Webapplikation meldet sich immer mit dem Benutzerkonto des gerade aktiven Nutzers bei *JUST* an. Es werden somit keine `root` Zugriffe auf die Daten benötigt und Sicherheitsrichtlinien müssen nicht geändert werden.

SK8: "Die Applikation soll sehr flexibel erreichbar sein."

Die Webapplikation ist über beliebige Browser von verschiedensten Endgeräten aus erreichbar und kann über das Internet weltweit aufgerufen werden.

Die optionalen Kannkriterien wurden ebenfalls erfolgreich umgesetzt:

KK1: "Die Applikation soll möglichst einfach und intuitiv bedienbar sein."

Die Oberfläche der Webapplikation ist möglichst einfach gehalten. Es wurden standardisierte Oberflächen-Komponenten, wie der Dateibaum, verwendet, um eine intuitive Darstellung zu erreichen. Der Nutzer kann alle Informationen auf einen Blick sehen und muss nicht durch viele Ansichten navigieren.

KK2: "Nutzer müssen nicht zwingend ein Benutzerkonto auf den HPC-Systemen besitzen."

Die Webapplikation wurde auch für Nutzer konzipiert, die keinen eigenes Benutzerkonto auf *JUST* besitzen. Solche Nutzer können nur die ihnen freigegeben Daten sehen, bekommen aber kein eigenes Heimatverzeichnis.

KK3: "Die Applikation soll leicht erweiterbar sein."

Die Software wurde modular entwickelt. Durch das *MVC*-Muster können Funktionalitäten leicht erweitert und ausgetauscht werden. Das Konzept wurde durch klare Schnittstellen erweiterbar gestaltet. Es können z.B. bei Bedarf Nutzergruppen, durch das Interface `Target`, implementiert werden, denen dann Dateien freigegeben werden können.

6.3 Fallbeispiel: Freigabe einer Datei

Im Rahmen des *Human Brain Project* möchte das Projektmitglied „Martin“ sein Projektergebnis mit einem anderen Projektmitglied „Andreas“ austauschen.

„Martin“ hat im Laufe des Projektes auf *JUQUEEN* gerechnet und seine Ergebnisse in seinem Heimatverzeichnis auf *JUST* in der Datei „`ergebnisse.h5`“ abgespeichert. Nun möchte er diese Datei seinem Projektpartner „Andreas“ zum Lesen freigeben.

„Andreas“ besitzt kein Benutzerkonto im *LDAP*-Server des *JSC* und hat somit keine Dateien auf *JUST*. Weiterhin ist „Andreas“ kein Mitarbeiter des *FZJ* und es besteht für ihn keine Möglichkeit sich ins interne Netz zu verbinden.

Bisher bestand für die beiden Projektpartner noch keine Lösung, wie sie die Datei austauschen können. Durch die im Rahmen dieser Arbeit entwickelten Webapplikation ist dies jedoch jetzt möglich.

Der Nutzer „Martin“ muss sich zunächst gegenüber der Webapplikation authentisieren und bekommt anschließend seine Dateien angezeigt.

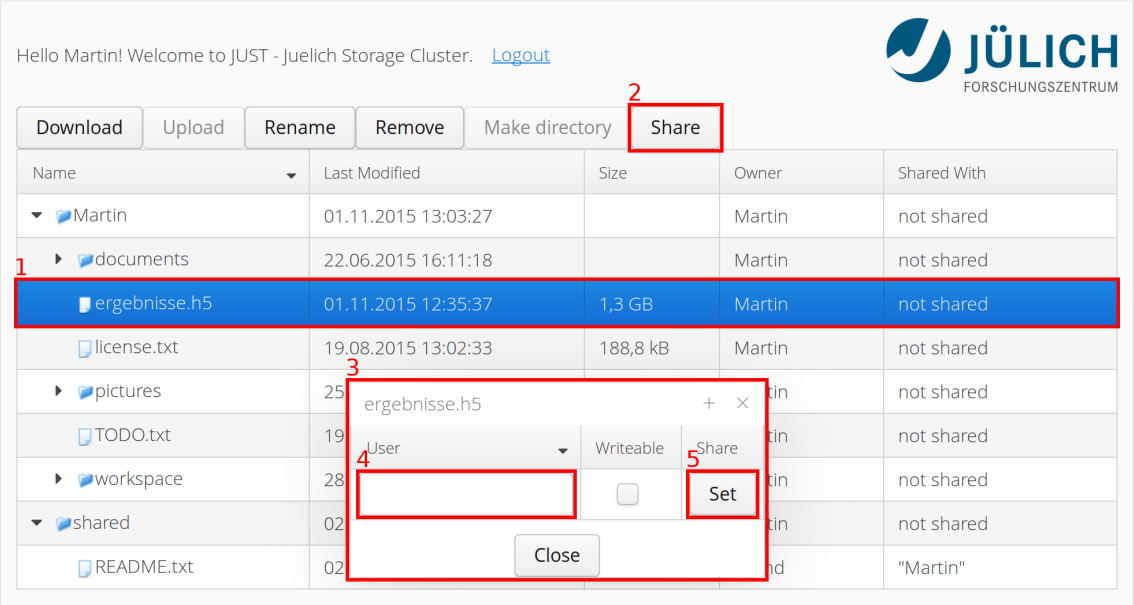

Abbildung 6.1: Freigabe einer Datei

Die Abbildung 6.1 zeigt die Oberfläche der Webapplikation nach erfolgreicher Authentisierung des Nutzers „Martin“. Die Datei „`ergebnisse.h5`“ liegt im Heimatverzeichnis des Nutzers.

1. Zunächst wählt „Martin“ die Datei im Dateibaum aus, die ausgewählte Zeile wird hierbei blau hinterlegt. Anhand des Eintrags in der Spalte „Shared With“ erkennt

„Martin“, dass die Datei zur Zeit noch keiner anderen Person freigegeben wurde.

2. Zur Freigabe klickt „Martin“ anschließend auf die Schaltfläche „Share“.

3. Daraufhin öffnet sich ein neues Fenster, in dem alle Freigaben für die Datei verwaltet werden können.

4. Um die Datei nun seinem Projektpartner freizugeben, gibt der Nutzer „Martin“ den Nutzernamen, in diesem Fall „Andreas“, in das Textfeld ein.

5. Mit einem Klick auf die Schaltfläche „Set“ wird die Freigabe aktiviert. Da das Kontrollkästchen in der Spalte „Writeable“ nicht gesetzt ist, wird die Datei nur zum Lesen freigegeben.

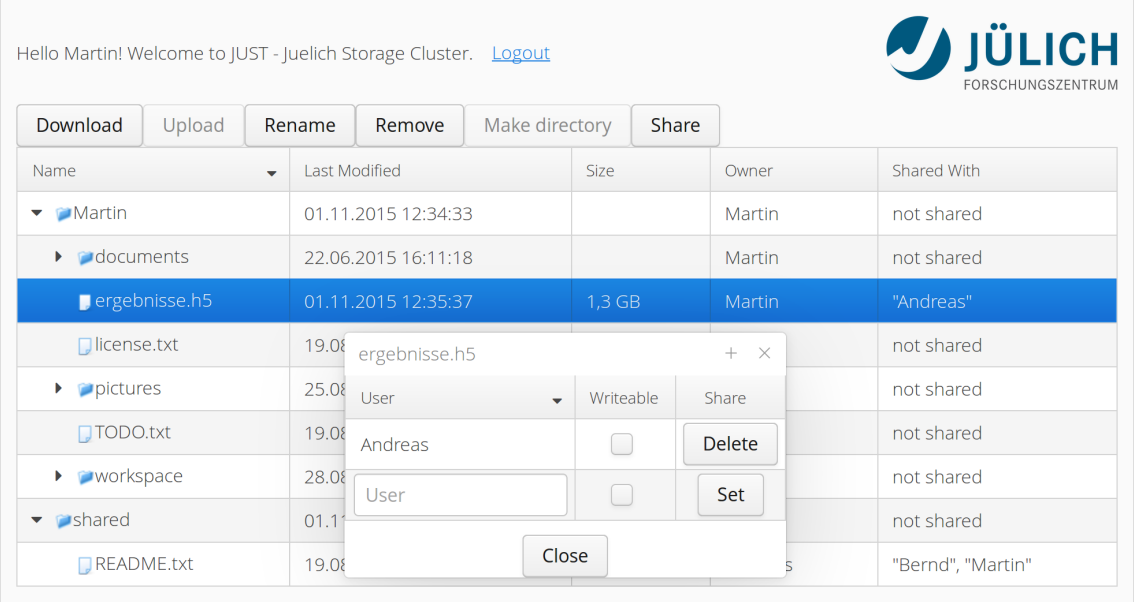

Abbildung 6.2: Freigabe beim Nutzer „Martin“ gesetzt

Die Datei ist nun dem Nutzer „Andreas“ freigegeben. In der Tabelle des Freigabe-Fensters wird eine neue Zeile hinzugefügt (siehe Abbildung 6.2). In dieser Zeile steht nun der Nutzer „Andreas“. Hier kann die Freigabe jederzeit wieder, durch Betätigung der Schaltfläche „Delete“, aufgehoben werden. Die Freigabe kann auch über das Kontrollkästchen „Writeable“ um Schreibrechte erweitert werden. Andererseits erscheint der Name des Nutzers „Andreas“ auch im Dateibaum in der Spalte „Shared With“ (siehe Abbildung 6.2).

Wenn sich der Nutzer „Andreas“, nachdem ihm die Datei freigegeben wurde, an der Applikation anmeldet, sieht er die Datei „`ergebnisse.h5`“ in seinem „shared“-Ordner (siehe Abbildung 6.3).

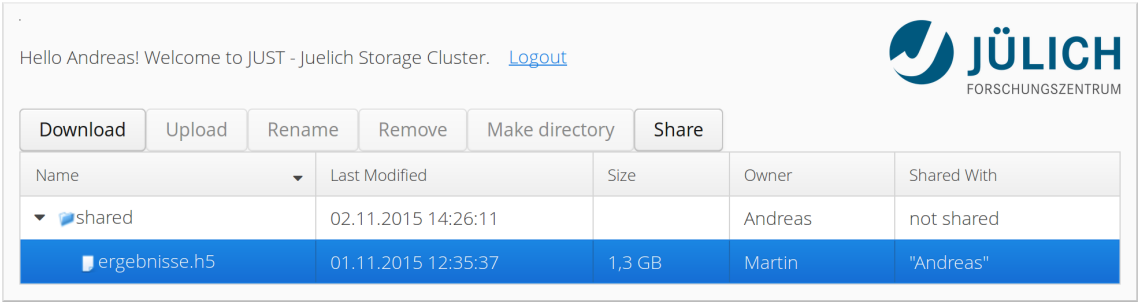

Abbildung 6.3: Freigabe beim Nutzer „Andreas“ gesetzt

Da „Andreas“ jedoch kein Benutzerkonto im *LDAP*-Server besitzt, sieht er kein Heimatverzeichnis. Zudem erkennt man, dass bei Auswahl der Datei nur die beiden Schaltflächen „Download“ und „Share“ aktiviert sind. Dies liegt daran, dass „Martin“ die Datei nur mit Leserechten freigegeben hat.

„Andreas“ kann die Datei nun weltweit herunterladen und die Ergebnisse von „Martin“ lokal auf seinem Endgerät betrachten bzw. auswerten. Außerdem könnte er über die Schaltfläche „Share“ die Freigabe beenden.

Dieser Fallbeispiel macht deutlich, dass ein schneller und intuitiver Datenaustausch im *HPC*-Umfeld nun möglich ist. Der Prozess hat nun eine klare Struktur bekommen. Die Freigabe ist durch die Verschlüsselung der Übertragungen sicher, außerdem vertrauenswürdig, da einzelne Dateien freigegeben werden können, ohne dass die Sicherheit anderer gefährdet wird.

6.4 Analyse verbleibender Angriffsszenarien

Die Applikation erfüllt durch verschiedene Sicherheitsmaßnahmen einen hohen Sicherheitsstandard (siehe Kapitel 4.4). In diesem Abschnitt werden die verbleibenden Risiken und mögliche weitere Verbesserungen der Sicherheit diskutiert.

Schafft es ein Angreifer die Shell zu übernehmen, in der die Webanwendung läuft, verschafft er sich Zugriff auf das gesamte Benutzerkonto, unter welchem die Webanwendung gestartet wurde. Solch ein Angriff wäre dadurch möglich, dass neue Sicherheitslücken in den verwendeten Komponenten der Software gefunden werden, z.B. in *Java*, *OpenSSL* oder in anderen Bibliotheken auf dem System. Auch wenn ein Angreifer es schafft, direkten Zugriff auf die Hardware des Servers zu erhalten, kann eine Übernahme der Shell stattfinden. Er kann dann auf alle Daten der Webapplikation, also insbesondere auch auf den privaten *SSH*-Schlüssel, zugreifen und sich damit beim *UFTPD* anmelden. Dieser glaubt nun er würde mit der ihm vertrauten Webanwendung kommunizieren, also erlaubt er dem Angreifer sich unter jeder User-ID aller Nutzer anzumelden. Somit kann der Angreifer auf alle Nutzerdaten frei zugreifen, welche auf den Dateisystemen liegen. Er könnte Daten löschen,

manipulieren oder auch sensible Daten auslesen.

Um einem solchen Angriff entgegenzuwirken, werden die *SSH*-Schlüssel meistens mit einem Passwort geschützt, das jedoch nicht auf dem Webserver gespeichert werden darf, da ein Angreifer nach Übernahme des Systems auf den Schlüssel zugreifen könnte. Der Nachteil ist nun aber, dass ein Administrator beim Starten der Webanwendung immer dieses Passwort eingeben muss.

Schafft es ein Angreifer trotz der Validierung aller Eingaben, den Code der Webanwendung zu verändern, ob durch das Injizieren von Code oder auf einem anderen Weg, hätte dies ähnliche Konsequenzen wie im obigen Szenario. Der Angreifer könnte die Webanwendung ebenenfalls komplett übernehmen und wieder auf alle Nutzerdaten zugreifen. Hierzu wären umfassende Kenntnisse in der *Java*-Programmierung sowie im Umgang mit dem *UFTP*-Protokoll erforderlich.

Wenn jedoch verhindert würde, dass die Webapplikation Zugriff auf alle Daten aller Nutzer bekommt, könnte kein Angreifer fremde Rechte missbrauchen. Der Zugriff auf die Daten der Nutzer könnte dadurch beschränkt werden, dass er z.B. nur über eine gültige *SAML* „assertion“ genehmigt wird.

Das heißt, der *UFTPD* würde den Datenzugriff nur erlauben, wenn er eine gültige *SAML* „assertion“ bekommt, welche die Authentifizierung des Nutzers bestätigt. Diese wird von einer vertrauenswürdigen Stelle, in diesem Fall von *Unity*, signiert. Dadurch wüsste der *UFTPD* jederzeit welcher Nutzer gerade auf die Daten zugreift und könnte die Zugriffsrechte entsprechend regulieren. Diese „assertion“ dürfte nur für kurze Zeit gültig sein, damit ein Angreifer sie nicht auslesen und missbrauchen kann.

Um einem Nutzer den Zugriff auf freigegebene Dateien eines anderen Nutzers zu gewährleisten, müsste der *UFTPD* zusätzlich eine *SAML* „assertion“ erwarten, die diesen Zugriff autorisiert. Diese kann langlebig sein und ist nur zusammen mit der ersten „assertion“ gültig, welche die Authentifizierung bestätigt. Sobald ein Nutzer einem anderen Nutzer eine Datei freigibt, müsste die Webapplikation eine „assertion“ erstellen, welche die Autorisierung validiert.

Die Webapplikation darf diese aber nicht selbst signieren. Sonst könnte ein Angreifer sich nach Übernahme des Webservers auch selber autorisieren um auf Dateien anderer Nutzer zuzugreifen. Also muss die Webapplikation sich die „assertion“ von einer dritten Stelle signieren lassen, wie z.B. *Unity*. Um sicherzustellen, dass die Anfrage nicht von einem Angreifer gestellt wurde, müsste *Unity* sich die Autorisierung wiederum von dem Besitzer der Daten bestätigen lassen. Der Nutzer müsste die Autorisierung also erneut, durch z.B. Eingabe von Nutzername und Passwort, erlauben. Anschließend könnte *Unity* die „assertion“ signieren und es wäre sichergestellt, dass nur der Besitzer einer Datei diese freigeben kann.

Da nun aber *Unity* alle „assertions“ signieren würde, verschiebt sich das Sicherheitsrisiko dorthin. Falls ein Angreifer den *Unity*-Server übernimmt, könnte er sich selbst „assertions“ ausstellen und signieren. Er könnte sich also wieder Zugriff auf alle Daten verschaffen.

Die höchste Sicherheit wäre gewährleistet, wenn Nutzer ihre Freigaben selbst signieren würden. Hierzu würde jedoch jeder Nutzer ein eigenes Zertifikat benötigen. Der Nutzer würde die Freigabe dann mit seinem privaten Schlüssel signieren und sich gegenüber dem *UFTPD* validieren. Dadurch wäre sicher gestellt, dass nur der Besitzer Freigaben auf seine Daten setzen kann.

Diese Maßnahmen würden die Webanwendung zwar sicherer machen, jedoch auf Kosten der Nutzerfreundlichkeit. Es ist nicht praktikabel von jedem Nutzer ein eigenes Zertifikat zu verlangen, da nicht jeder Nutzer der *HPC*-Systeme ausreichend technisch versiert ist um ein eigenes Zertifikat in sein System einzubinden und den privaten Schlüssel auch vor Angreifern zu schützen. Zudem würde dies einen enormen verwaltungstechnischen Aufwand erzeugen. Wenn hingegen *Unity* die *SAML* „assertion“ signiert, muss der Nutzer jede Freigabe durch eine erneute Authentisierung bestätigen. Dies wäre für den Nutzer sehr umständlich und ebenfalls nicht Benutzerfreundlich.

Der *UFTPD* müsste noch um die Funktionalität, mit *SAML* „assertions“ umgehen zu können, erweitert werden. Außerdem müsste auch *Unity* um Funktionalitäten erweitert werden.

Eine absolute Sicherheit kann zum aktuellen Stand nicht gewährleistet werden. Durch eine entsprechende Anpassung der genannten Funktionalitäten könnte die Sicherheit des Projektes nochmals erhöht werden.

6.5 Zusammenfassung

Im Rahmen dieser Masterarbeit wurde eine sichere, webbasierte Datenaustauschlösung im HPC-Umfeld konzipiert und entwickelt. Über eine Webapplikation können die Nutzer auf ihre Daten zugreifen und diese mit anderen Nutzern austauschen. Projekte können nun viel effizienter durchgeführt werden. Die Webapplikation bietet den Projektgruppen jetzt die Möglichkeit, ihre Daten weltweit, auf eine einfache und intuitive Art, sicher miteinander auszutauschen und zu bearbeiten.

Die Oberfläche wurde mit *Vaadin* entwickelt und ist über einen bliebigen Browser über das Internet verfügbar. Dies hat den großen Vorteil, dass mit der Applikation auch mobil, z.B. per Smartphone bzw. Tablet, gearbeitet werden kann. Die grafische Oberfläche ist intuitiv gestaltet, leicht zu bedienen und selbsterklärend. Durch die Gliederung in Form der *TreeTable* werden die Daten übersichtlich dargestellt. Auf einen Blick bekommt ein Nutzer eine Übersicht über die ihm freigegebenen Daten. Es müssen keine Konsolenbefehle verwendet werden, sondern alle Funktionen sind mit der Maus steuerbar.

Der Zugriff auf die Dateisysteme geschieht über das sichere *UFTP*. Die Applikation hat als einzige Voraussetzung, dass auf dem Server der *UFTPD* läuft. Dadurch ist die Webapplikation nicht an ein bestimmtes Dateisystem gebunden und kann auch in anderen Umgebungen eingesetzt werden. Weiterhin müssen Sicherheitsrichtlinien nicht verändert werden, hierdurch wird der administrative Aufwand gering gehalten.

Durch die Webapplikation wurde ein einheitlicher Datenzugriffspunkt geschaffen. Die Nutzer können zielgerichtet Daten freigeben und der Datenaustausch ist reguliert. Externe Nutzer können somit die Plattform nicht für einen illegalen Datenaustausch missbrauchen.

Da die Daten auf den vorhandenen Dateisystemen bleiben, entsteht kein zusätzlicher Datenverkehr durch Synchronisation verschiedener Server. Es werden auch keine weiteren Speicherkapazitäten benötigt, wie dies z.B. bei einer Lösung mittels einem *FTP*-Server erforderlich gewesen wäre. Weiterhin sind der Speicherort der Daten und die berechtigten Nutzer jederzeit bekannt. Die Daten bleiben, wie vom *JSC* gefordert, im Gegensatz zu herkömmlichen Cloud Diensten, im eigenen Besitz.

Die einzelnen Komponenten und verwendeten Protokolle sind genauestens bekannt, da die Webapplikation eigenständig entwickelt wurde. Dadurch entstehen mehrere Vorteile, z.B. kann Angriffen besser entgegen gewirkt werden und die Software ist, ohne vorherige langwierige Einarbeitung, leicht erweiterbar. Durch die Benutzung bereits vorhandener Ressourcen wird der administrative Aufwand noch weiter minimiert. Klare Entwurfsmuster, einheitliche Namensgebungen und strukturierte Schnittstellen machen die Software modular und sehr flexibel. Bei der Entwicklung des Programms wurde zunächst darauf geachtet, keine unnötigen Funktionen zu implementieren. Es wurde jedoch viel Wert auf die Erweiterbarkeit des funktionalen Umfangs gelegt. Somit können später weitere Funktionen sehr leicht hinzugefügt werden.

Durch die Nutzerverwaltung mittels *Unity* können auch Personen, die kein eigenes *LDAP*-Benutzerkonto für *JUST* besitzen, die Webapplikation nutzen. Hierbei müssen keine temporären Benutzerkonten eingerichtet werden.

Die Webapplikation wurde bereits externen Interessenten vorgestellt. Die Resonanz fiel durchweg positiv aus und ein Mehrwert wurde erkannt. Gerade das schlichte Design, keine Überhäufung des Funktionumfangs, sowie die Sicherheit und die Unabhängigkeit vom Dateisystem überzeugten die Interessenten von dem Projekt.

6.6 Ausblick

Zur Zeit wird der Nutzer noch von der Webapplikation selbst authentifiziert. Im nächsten Schritt wird die Applikation, wie bereits konzipiert, durch *Unity* für alle Mitglieder des DFN-AAI verfügbar gemacht. Hierfür sind bereits alle Schnittstellen vorhanden.

In einem weiteren Schritt soll es möglich werden, Dateien über eine eindeutige *URL* anderen Personen zur Verfügung zu stellen. Dies ist durch das Interface `Target` bereits vorgesehen. Somit könnten Dateien auch anonymen, bzw. nicht angemeldeten Nutzer zur Verfügung gestellt werden.

Bisher können nur einzelne Dateien heruntergeladen werden. In Zukunft wäre es auch denkbar, dass Nutzer komplette Verzeichnisse herunterladen können. Die Verzeichnisse würden hierbei zunächst als Archiv gepackt und anschließend zum Herunterladen zur Verfügung

gestellt. Diese Lösung wäre jedoch nur für Verzeichnisse bis zu einer bestimmten Größe praktikabel.

Zur Zeit können Dateien nur einzelnen Nutzern zur Verfügung gestellt werden. Durch das Interface `Target` ist es möglich, die Applikation zu erweitern, so dass Freigaben auch für ganze Nutzergruppen gesetzt werden können.

Weiterhin wäre es denkbar Dateien direkt in der Weboberfläche als Vorschau zu öffnen. Eine *PDF*-Datei könnte zum Beispiel im Browser internen *PDF*-Plugin angezeigt werden.

Diese Funktionalitäten können durch die strukturierte Programmierung leicht implementiert werden. Jedoch muss darauf geachtet werden, dass der Funktionsumfang nicht überladen wird, so dass die Simplizität und die Sicherheit der Applikation auf keinen Fall beeinträchtigt werden.

Literaturverzeichnis

[1] Oliver Bücker u. a. „Towards a Multiscale, High-Resolution Model of the Human Brain“. In: *Brain-Inspired Computing*. Springer, 2014, S. 3–14.

[2] Gerald Carter. *LDAP system administration*. O'Reilly Media, Inc., 2003.

[3] IBM Corporation. *General Parallel File System Version 3 Release 5.0-3 Concepts, Planning, and Installation Guide*. 2012.

[4] Joshua Davies. *Implementing SSL/TLS using cryptography and PKI*. John Wiley und Sons, 2011.

[5] *DFN-AAI - Authentifikations- und Autorisierungs-Infrastruktur*. URL: https://www.aai.dfn.de/ (besucht am 21. 08. 2015).

[6] *Forschungszentrum Jülich*. URL: https://www.fz-juelich.de/ (besucht am 13. 07. 2015).

[7] Erich Gamma u. a. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[8] Marko Grönroos. *Book of Vaadin*. Vaadin Ltd, 2015.

[9] Frédéric Magoulès. *Fundamentals of grid computing: theory, algorithms and technologies*. CRC Press, 2009.

[10] James Martin und Joe Leben. *TCP-IP-Netzwerke: Architektur, Administration und Programmierung*. Prentice Hall, 1994.

[11] David Remy und Jothy Rosenberg. *Securing web services with WS-security*. Sams Publishing, Carmel, IN, 2004.

[12] TOP500. *SUPERCOMPUTER SITES*. URL: http://www.top500.org/lists/2015/11/ (besucht am 20. 11. 2015).

[13] *UNICORE*. URL: https://www.unicore.eu/ (besucht am 02. 07. 2015).

[14] *UNICORE UFTPD server*. URL: https://www.unicore.eu/documentation/manuals/unicore6/files/uftp/uftp-manual.html (besucht am 12. 07. 2015).

[15] *Unity*. URL: http://unity-idm.eu/site/ (besucht am 01. 08. 2015).